



Angular Series 9-Part Tutorial



Table of Contents

1. ToC
2. Additional Angular Resources
3. Introduction
4. Getting Started with the Angular CLI
5. Application Structure Using Modules
6. Setting Up Routing in Your Application
7. Inputs, Outputs, and EventEmitters
8. Angular Component Lifecycle
9. Template Driven Angular Forms
10. Cover
11. Getting Started with Electron and Angular CLI
12. Using Bootstrap 4 with Angular

SERIOUS ABOUT ENSURING YOUR ANGULAR PROJECT IS ON THE RIGHT TRACK?

Take Advantage of Our Free One-Hour Project Consultation Offer

Serious about ensuring your Angular project is on the right track? For a limited time, Mark Root, the author of the Angular Tutorial Series, will be offering short one-hour consults.

Limited available slots. Visit the link below to learn what you can expect and to reserve your spot.



[SIGN UP TODAY](#)

Introduction



Looking to get yourself or your team up to speed with Angular? Having trained many teams on the basics of Angular, Intertech developer Mark Root has compiled all of his materials into one comprehensive guide. Designed to walk developers through basic concepts as well as some more advance topics, this 9-part series is great starting point for anyone wishing to become more proficient with Angular.

About Intertech

Founded in 1991, Intertech delivers technology [training](#) and [software development consulting](#) to Fortune 500, Government and Leading Technology institutions.

Learn more about us . Whether you are a developer interested in working for a company that invests in its employees or a company looking to partner with a team of technology leaders who provide solutions, mentor staff and add true business value, we'd like to meet you.

www.intertech.com

Chapter 1: Getting started with the Angular CLI

I have worked with Angular and the Angular CLI since Angular's release candidate days. It has come a long way since then. New releases of the framework no longer take a laborious process to upgrade. I recently upgraded a project from v6 to v7 when v7 came out and it was painless and almost unnoticeable. Many of the changes were under the hood. Personally, that was exciting, especially having upgraded Angular projects through all of Angular's previous versions. Angular has gotten to the point as a framework that new releases are primarily for new features and bug fixes. That is music to a developer's ears! Just scanning the Angular CHANGELOG.md, the number of breaking changes since the release of v4.0 has dropped dramatically. What you learn today when developing with Angular (and from this Angular tutorial) shouldn't change too much down the road and that hasn't always been the case.

As a consultant, I have been on a few projects over the last years that have revolved around getting developers up to speed on developing with Angular v2+. Over that time there has been a consistent group of content that I have had to cover when teaching the basics of Angular. What I hope to do is create a series of Angular tutorials that cover that material. To do that I am going to generate an example Angular project and add different example pieces of Angular code that I will use to demonstrate pieces of Angular's framework. I intend to use it as my personal toolbox (as well as these posts on our blog) as a place I can reference when I need to remember how to do certain things in Angular.

That being said, the first thing we should explore is the Angular CLI which we'll do in this . If you aren't familiar with the Angular CLI, it's an installable command line interface that can be used to generate Angular projects, components, services, and modules. It's an extremely useful tool to use when developing Angular projects. Through a few commands a developer can get up on an Angular project with little effort. That's just what this Angular tutorial is going to do. I'll explain a bit about what is generated as well.

The repository can be found at <https://github.com/rmroot/ng-example>.

Dependencies

Before we get started with this Angular tutorial there are a few development environment dependencies that will be needed. Here's what you will need to develop with the Angular CLI:

- Node.js: v6.9.x +
- : 3.x.x +
- The Angular CLI
 - To install, open a terminal with installed and then run
 - `install -g @angular/`

Creating and Running the Application

Once the Angular CLI is installed. Run the following command to create a new application. In my case, I named my application "ng-example".

- `ng new ng-example`

Once your application finishes generating, go into your project directory and start your application with the following command:

- `ng serve`

The application will build and be served up to Navigate to that URL in your browser and you will see the default application shell provided by the CLI.

The Angular CLI Application Structure

Before we start developing, let's take a look at what the CLI generated for us. Open the project in your favorite IDE; I encourage you to use VS Code. VS Code's intelli-sense for typescript is very powerful, as well as its ability to help with imports. It also has a number of Angular specific plugins for developers to use that are very helpful.

When you look inside your project folder, you will see that the CLI has generated a bunch of stuff for you. I am not going to dive into every detail in this post, but I do want to highlight a few things that you should know to get started. The CLI team wants to provide its users the ability to start development right out of the box and they have done just that. Inside the `src/` you'll find everything you need to get started. Here is a quick rundown:

src/*

- `styles.css`

Opening this file you see the comment “You can add global styles to this file, and also import other style files”. That pretty much sums that up, there are a number of strategies for styling your application. For anything global, put it in this file or create a new `.css` file and import it here. This will also be where you can import Bootstrap or any other styling frameworks you wish to add.

- `index.html`

This is where the application gets started. Pretty typical here with the interesting bit being the `<app-root></app-root>` tag. This is a reference to your entry component in your application (`AppComponent`). I'll explain how that tag works in a moment.

src/app/*

- `app.component.ts`

This file contains your application's first and only component right now. It's a pretty bare bones component for the time being but I want to highlight one thing:

```
1 @Component({
2   selector: 'app-root',
3   templateUrl: './app.component.html',
4   styleUrls: ['./app.component.css']
5 })
```

This is what marks a typical class file as an Angular component. There's a variety of stuff that can be put in this section that I will cover in a later blog post, but for now let's look at the three default things that go in any component.

1. *selector: 'app-root'* - Here is the `app-root` that we saw earlier. This is what tells the application that `<app-root>` means to load up this component (`AppComponent`).

2. *templateUrl: './app.component.html'* - This tells your application what template or view to use in association with this component. In this it is using `app.component.html` was also generated by the Angular CLI.

3. *styleUrls: ['./app.component.css']* - This is providing the component with its own style sheet. Any style you want for just your `AppComponent`, put it in the `app.component.css`. You may notice that it is an array. This means you can declare multiple `styleUrls`. This can be useful if you want to reuse styles between just a couple of components

- `app.module.ts`

Angular uses modules to manage your components. I am not going to go into full detail of how this works in this chapter but here is a quick rundown of what's happening here.

`@NgModule` is similar to `@Component` in that it is marking this class file as a module.

Inside it, we are declaring our `AppComponent`, every component in your application needs declaration in a module.

Summary

With that we have generated an Angular project using the Angular CLI. I have provided you a brief explanation of what the CLI generated for us and how some of it works. In the next chapter, we will discuss application structure using modules.

Chapter 2: Application Structure Using Modules

In the previous chapter, we generated a project using [CLI](#). In this I am going to discuss the structure of an Angular application. Specifically focusing on using Angular modules to manage your code base. I will start by explaining what an Angular module is and how they work. Then I will begin to lay the for our application's structure.

What is an Angular Module?

I like to think of an Angular module as a declaration of everything that will be used or needed in a section of the application. For example, let's take a look at what is inside of our AppModule. As you may recall, the AppModule was generated for us by the CLI and contains everything we need for a working application shell:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 @NgModule({
5   declarations: [
6     AppComponent
7   ],
8   imports: [
9     BrowserModule
10  ],
11  providers: [],
12  bootstrap: [AppComponent]
13 })
14 export class AppModule { }
```

So how does this Angular module work with my statement “a module is a declaration of everything that will be used or needed in a section of the application”? Well, first we need to decide what “section” of the application this is. Since it's the AppModule, the section we are declaring here is the entire application.

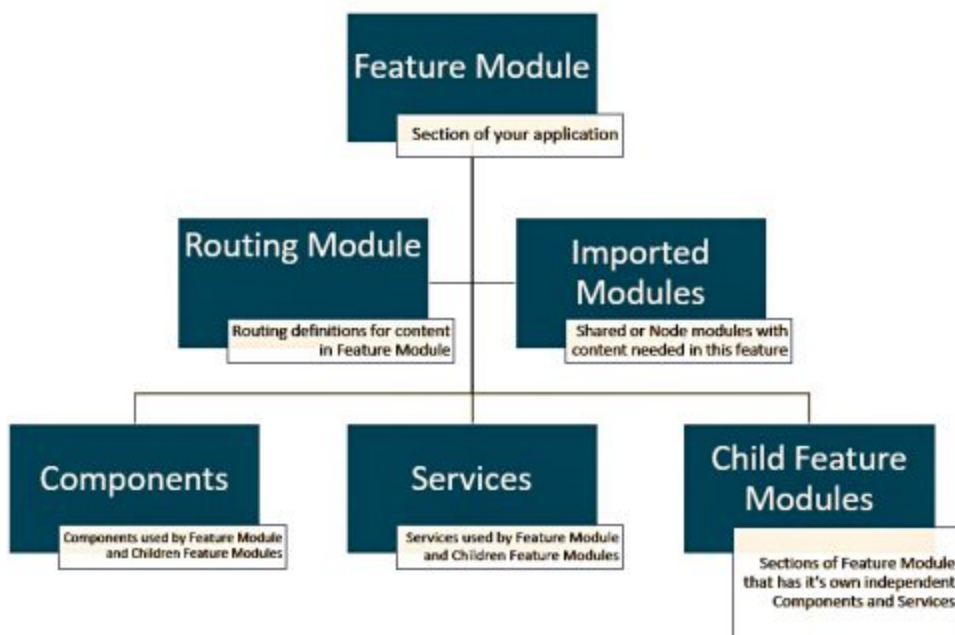
So then what do we have declared for this section? What are we telling our entire app that it needs to use? Well, we are literally declaring our AppComponent, which is necessary to use it. Every component in your application needs to be declared in an Angular module to be able to use it. Additionally, we have imported a BrowserModule. The browser module is the “ng module for the browser” according to Angular's documentation. It's essentially the middleman between the browser and our app.

*If you are wondering about the bootstrap array, look in main.ts and notice the AppModule being provided in the application bootstrapping method. AppComponent in that array is telling the bootstrapping method what component to bootstrap. You will also notice that the platformBrowserDynamic object is imported from the same place as our BrowserModule. Had we not imported the BrowserModule in our AppModule we wouldn't be able to use that object because our app wouldn't know about it.

With that, you can see, we have declared our essentials for our application in our AppModule. As the application grows, additional things may need to be declared but for we have everything we need for this "section" (the whole app).

Angular Module Structure

When setting up modules in an Angular application, I like to keep this hierarchy picture in mind.



Child feature modules will hold the same hierarchy. If your able to maintain this structure your application will be much easier to maintain and separation of concerns will be easier to implement. Ideally, this tree will grow wider faster then it grows deeper, meaning your child feature modules will be independent of a parent module.

Feature, Shared, and Core Modules

Now depending on the size and complexity of the project, you may be able to just use the single `AppModule` for all of your declarations. However, for larger applications, I encourage the use of core, shared and feature modules. Doing so will ensure proper separation of concerns, which will ease scalability as your application grows. I'll try to briefly explain how this works by describing each type of module.

- Feature Modules
 - A feature module is a module in which all of the content is going to be encapsulated inside of a single area.
 - Applications should be made up of multiple feature modules.
 - Think of a feature module as a mini application inside your full application.
 - A feature would be what I mean by "section", it usually has a root component that it exports and is used in a parent module. All the rest of the pieces of that feature will be put inside that root component.
- Shared Modules
 - Use shared modules for pieces of your application that need to be used across multiple areas (features) of your application.
 - If a component is going to be re-used in multiple features, declare it in a shared module.
 - Services and Pipes are more commonly declared in shared modules.
 - Shared modules don't necessarily follow the "section" idea that I stated previously. they provide a way to share common pieces to fill out feature module "sections".
- Core Module
 - I prefer to use a core module as a way to separate the configuration layer of your application with the rest of your application.
 - To do so, you declare all of your feature and shared modules in your `CoreModule` and just provide `CoreModule` to your `AppModule`
 - For anything that needs to be used across all feature modules, declare it in the `CoreModule`.
 - Think of it as the parent feature module for all of the content you are going to add to your application.

Setting Up This Structure in our Application

I know that may not be totally clear, it's a difficult concept to relay through writing. So I'll try to demonstrate through practice. Let's set up our application so that we can use this Angular module structure and hopefully that will make things clearer. I will create a new feature for every "example" I add to the repository. Before I start adding those examples, I have to set up the to use this design.

First, let's generate our core module, core component and a shared module. To do so, execute the following commands:

- `ng g module core`
- `ng g component core`
 - If you do the module first, a folder will be created with the core module. Then, when the component is generated, it will be placed in the same folder and be declared in the module by the CLI.
- `ng g module shared`
 - We aren't going to do anything with `SharedModule` but I like to get it in early for future use.

Next, we need to tell `AppModule` about our `CoreModule`

- Import the `CoreModule` in `AppModule` and add it to the imports array
- `AppModule` will now look like this

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { AppComponent } from './app.component';
4 import { CoreModule } from './core/core.module';
5 @NgModule({
6   declarations: [
7     AppComponent
8   ],
9   imports: [
10    BrowserModule,
11    CoreModule
12  ],
13  providers: [],
14  bootstrap: [AppComponent]
15 })
16 export class AppModule { }
```

- Next open `app.component.html` and replace the current content with `app-core`
- Reminder: "app-core" is the selector for the core component, so all we are doing is telling `AppComponent` to serve up `CoreComponent`

If you run `ng serve` and navigate to `localhost:4200` you will notice there is an error in the console `"'app-core' is not a known element"` and nothing displayed on the page. This is telling us that our application isn't aware of a component with a selector of `"app-core"`. Why is that? Well, we declared `CoreComponent` in our `CoreModule` but we didn't expose it our app. To do so, add the `CoreComponent` to the exports array in our `CoreModule`. Your `CoreModule` should look like this:

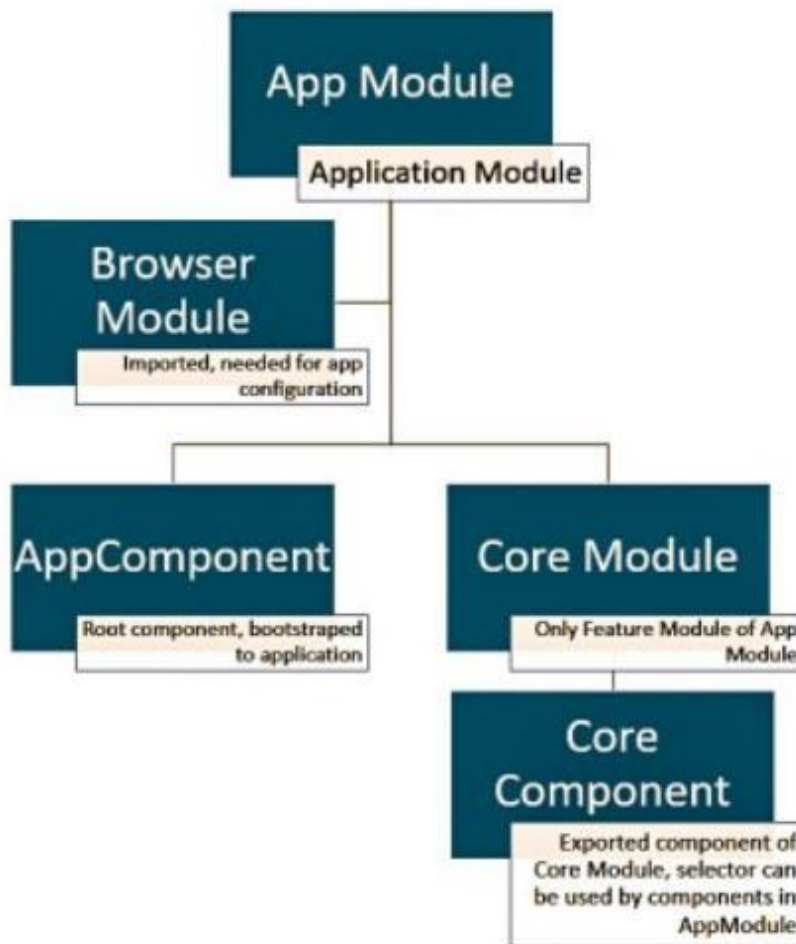
```
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { CoreComponent } from './core.component';
4
5 @NgModule({
6   imports: [
7     CommonModule,
8   ],
9   declarations: [CoreComponent],
10  exports: [CoreComponent]
11 })
12 export class CoreModule { }
```

Now when you run `ng serve` and check your browser, you will see `"core works!"`, which is the content in our template of our `CoreComponent`.

****Using a single exported component and selector is one way to set up Angular modules in an application. This method will allow a developer to create an entire application without using the router. In a later chapter, we will set up routing for our application and will not need to use the selector or the exports array.**

Summary

Let's take a look at the current hierarchy of our application.



From here we are ready to start adding features to our application. When adding features, you will follow the same pattern as we just did to add the CoreModule but instead, for additional features, we will be telling our CoreModule about our additionalFeatureModules. Next thing we are going to do is set up routing in our application, meaning we won't need to export our components for parent modules to use, instead we will use the router.

Chapter 3: Setting Up Routing in Your Application

In the previous chapter, I discussed application structure using [Angular modules](#), and set up the groundwork for using core, shared and feature modules. In this I will demonstrate how to basic routing in an Angular application using Angular Router Modules.

****Note:** I have added bootstrap to the application. I have written about adding bootstrap to an angular application in the final chapter of this tutorial.

So, let's get started with this Angular Router tutorial.

Generate Components

we are going to need some additional components in the application. Let's add a dashboard, sidebar banner:

- `ng g component core/dashboard`
- `ng g component core/sidebar`
- `ng g component core/banner`

These components should be added under the core/ folder and declared in our CoreModule. Since these are root components for our CoreModule we don't need to create a feature module for them. we are going to set up our application routing. We will come back to these components shortly.

Create and Setup AppRoutingModuleModule

The first thing we will have to do is add an AppRoutingModuleModule to define our root routes. In a previous we had placed our core component selector in our app.component.html. Instead of using the selector, we will setup routing and use routing to display our CoreComponent in app.component.html using a RouterOutlet

[Angular.io documentation](#) defines a RouterOutlet as *"a placeholder that Angular dynamically fills based on the current router state"*.

To do so, do the following:

1. If you are following along from previous chapters, `<app-core></app-core><router-outlet></router-outlet>` `app.component.html`. Otherwise replace the content in `app.component.html` with `<router-outlet></router-outlet>`.
2. Create a file `app-routing.module.ts` the same level as `AppComponent`
 - the future, when you create a feature module that is going to have routing. Use a `--routing` flag with the generate command and this file will be created for you.
3. Add the following code to `app-routing.module.ts`

```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { CoreComponent } from './core/core.component';
4 const routes: Routes = [
5   {
6     path: '',
7     component: CoreComponent
8   }, {
9     path: '**',
10    component: CoreComponent
11  }
12 ];
13 @NgModule({
14   imports: [RouterModule.forRoot(routes)],
15   exports: [RouterModule]
16 })
17 export class AppRoutingModule { }
```

4. Add the new `AppRoutingModule` to the imports array in `AppModule`

If you run `ng serve` you will notice that the application is the same as when we had `<app-core></app-core>` inside of `app.component.html`. The difference now is that `<router-outlet></router-outlet>` is dynamically selecting the `CoreComponent` based on the state of the Angular router. The `<router-outlet></router-outlet>` will check for a path corresponding to the routes provided to the in this routes, and display the component that goes with that path.

In this we have created two routes, one with the empty path and one with what is known as a **wildcard path**. Having `CoreComponent` with the empty path means that when you navigate to `localhost:4200` then you will be routed to `CoreComponent`. The wildcard path will match with any unrecognized paths. That means that any path we put after `localhost:4200` will display the `CoreComponent`. I would recommend adding some sort of path not found component and use that with the wildcard path but for now this will do.

`RouterModule.forRoot(routes)` is providing our defined routes to our application. All additional routes will have to be `forChild` as I will demonstrate below.

Adding Child Routes

we are going to add our content to our CoreComponent. This will demonstrate how child routes can be added and work with the `<router-outlet></router-outlet>`. We want to add the banner and sidebar and then have a content area with another `<router-outlet></router-outlet>` that our child route will navigate to. We will have it default to the `.`. Add the following code:

core.component.html

```
1 <div class="container-fluid">
2   <div class="row">
3     <div class="col-12">
4       <app-banner></app-banner>
5     </div>
6     <div class="col-2 sidebar">
7       <app-sidebar></app-sidebar>
8     </div>
9     <div class="col">
10      <router-outlet></router-outlet>
11    </div>
12  </div>
13 </div>
```

banner.component.html

```
1 <nav class="navbar navbar-dark navbar-expand">
2   <a class="navbar-brand">Angular.io Demo</a>
3 </nav>
```

sidebar.component.html

```
1 <div class="d-flex flex-column">
2   <div class="p-2">
3     Home
4   </div>
5   <div class="p-2">
6     Routing Example
7   </div>
8 </div>
```

I have added some basic bootstrap and styling classes to layout my components. As you can see, in our CoreComponent we added another `<router-outlet></router-outlet>` tag. We will have to add child route definitions for the CoreComponent route we previously defined. To do so we will create a new CoreRoutingModule in the core/ folder. As I stated earlier, using the `--routing` flag when generating the module will also generate the RoutingModule.

1. Create a `core-routing.module.ts` file in your core/ folder.
2. Add the following code

```

1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { DashboardComponent } from '../dashboard/dashboard.component';
4 import { CoreComponent } from '../core.component';
5 const coreRoutes: Routes = [
6   {
7     path: '',
8     component: CoreComponent,
9     children: [
10      {
11        path: '',
12        component: DashboardComponent
13      },
14      {
15        path: 'dashboard',
16        component: DashboardComponent
17      }
18    ]
19  }
20 ];
21 @NgModule({
22   imports: [RouterModule.forChild(coreRoutes)],
23   exports: [RouterModule]
24 })
25 export class CoreRoutingModule { }

```

3. Add CoreRoutingModule to the imports array of the CoreModule
4. Remove the empty path to the CoreComponent in the AppRoutingModule

What we are doing now is declaring the empty path route to CoreComponent in this module, and then giving child routes to that path. So now when we navigate to localhost:4200 we will get the CoreComponent and then in the <router-outlet></router-outlet> in core.component.html it will dynamically select our DashboardComponent. The second path for 'dashboard' means that we will get the same thing if we navigate to localhost:4200/dashboard.

I should point out that the child routes are declared for the empty path route to CoreComponent. Since we have the wildcard route used to get to CoreComponent as well, it should be noted that the child routes will not work with the wildcard route. Basically localhost:4200/somewildcard/dashboard is not going to display the same thing as localhost:4200/dashboard even though the wildcard navigates to the CoreComponent.

Depth First Search

A quick note on the functionality of the Angular router. When the Angular router looks through the application's defined routes for a corresponding route definition, it performs a search. For a quick demonstration of this, try the following.

Update your coreRoutes array to look like this:

```

1  const coreRoutes: Routes = [
2    {
3      path: '',
4      component: CoreComponent,
5      children: [
6        {
7          path: '',
8          component: DashboardComponent
9        },
10       {
11         path: 'dashboard',
12         component: DashboardComponent
13       }
14     ]
15   },
16   {
17     path: 'dashboard',
18     component: DashboardComponent
19   }
20 ];

```

If you navigate to localhost: the in AppComponent will match the empty string first, then search its children for 'dashboard' to go with the second RouterOutlet. It will find it and display the CoreComponent with the DashboardComponent in the router-outlet for CoreComponent. The second route to 'dashboard' here can never be found.

But if you change the array to look like this:

```

1  const coreRoutes: Routes = [
2    {
3      path: 'dashboard',
4      component: DashboardComponent
5    },
6    {
7      path: '',
8      component: CoreComponent,
9      children: [
10       {
11         path: '',
12         component: DashboardComponent
13       },
14       {
15         path: 'dashboard',
16         component: DashboardComponent
17       }
18     ]
19   }
20 ];

```

The Angular router in our AppComponent will see the 'dashboard' path before the empty path and just display the DashboardComponent without the CoreComponent.

To avoid confusion with the empty string as demonstrated above. I encourage you to use the option for your routes.

```

1     {
2       path: '',
3       pathMatch: 'full',
4       redirectTo: 'dashboard'
5     },
6     {
7       path: 'dashboard',
8       component: DashboardComponent
9     }

```

Now when we navigate to localhost:4200, the router-outlet in AppComponent will dynamically select CoreComponent and the router-outlet in CoreComponent will see the empty and redirect to localhost:4200/dashboard

routerLink Example

Assuming you will want to click around your application to navigate, you are going to need to understand the routerLink directive. Here's a quick example to get you started.

Since I want to treat my examples like features, I am going to create a RoutingExampleModule and a RoutingExampleComponent.

- `ng g module core/routing-example --routing`
 - Remember the routing flag will provide a routing module for future use.
- `ng g component core/routing-example`
- add `RoutingExampleModule` to the imports array in `CoreModule`
- Update `coreRoutes` in `CoreRoutingModule`
 - You will probably have to remove the DFS example stuff.

```

1 const coreRoutes: Routes = [
2   {
3     path: '',
4     component: CoreComponent,
5     children: [
6       {
7         path: '',
8         pathMatch: 'full',
9         redirectTo: 'dashboard'
10      },
11      {
12        path: 'dashboard',
13        component: DashboardComponent
14      },
15      {
16        path: 'routing-example',
17        component: RoutingExampleComponent
18      }
19    ]
20  }
21 ];

```

- `sidebar.component.html` use a directive to route to our components.

```
1 <div class="d-flex flex-column">
2   <div class="p-2">
3     <a routerLink="/dashboard" routerLinkActive="active">Home</a>
4   </div>
5   <div class="p-2">
6     <a routerLink="/routing-example" routerLinkActive="active">Routing Example</a>
7   </div>
8 </div>
```

Now when you run the application, clicking the links in the sidebar will change the content displayed by the `<router-outlet></router-outlet>` in `core.component.html`.

** `routerLinkActive="active"` will set the class "active" on the element when the Angular router matches that link. Any class can be set using this method:

`routerLinkActive="someClassForActive"`

Summary

With you should have all the tools you need to get started with application routing in Angular. In a future chapter, I will demonstrate additional functionality for the Angular router in the RoutingExample section, but for you should be able to get started with basic routing.

Chapter 4: Inputs, Outputs, and EventEmitters

In this chapter, I will discuss the basics of Angular components. I'll cover what makes up an Angular component, and give a brief demonstration using component I/O. The components we will be discussing will be generated using the CLI.

What is an Angular Component?

Angular components are the building blocks of any Angular application. A component is where the content of an application is placed. That means components provide the views or templates for your application, as well as the logic behind managing the data bound to those views. When using the Angular CLI to generate components (always use the CLI), you will be provided with four files for every component you generate. I briefly covered these files in my introduction to the CLI but will go over them again here. Below, I have pasted the piece of an Angular component's .ts file that will tell your application that this file is no ordinary typescript file, but rather an Angular component.

```
1 @Component({
2   selector: 'app-root',
3   templateUrl: './app.component.html',
4   styleUrls: ['./app.component.css']
5 })
```

This will look similar to an Angular module, but instead of `@NgModule`, we are using `@Component`. The behavior of the decorators is somewhat similar. By that I mean that the decorator is telling our application that this file is special. Furthermore, inside of the decorator, we are defining the pieces of this given component. Similarly, to the content inside of an `@NgModule` in which we declare the content inside of a given section of our application. This component comes from our application `AppComponent`. Let's take a look at what we are defining here.

- `selector: 'app-root'`
 - This is a tag that our application will use to select this component. Meaning that if we want to place this component somewhere within our application, all we have to do is add `<app-root></app-root>` inside the view of another component.
 - Two quick notes on that:
 - You wouldn't place `AppComponent` inside a different component, this is just for explanation purposes.
 - If you do want to use the selector to place your components inside other components, they will have to be known by a common module. I cover this in my [application routing](#) post.
- `templateUrl: './app.component.html'`
 - This is telling your application what template or view to render in association with this Angular component.
 - The CLI generates a html file for you when you create a new component but, if you want, you can write your html in this section using back-ticks. I don't recommend it but it doesn't hurt to know.
- `styleUrls: ['./app.component.css']`
 - This defines a style sheet. The CLI also generates this file for every component you create. If you want to add a style definition that is specific to a given component, add it to that component's style sheet.
 - You may notice that it is an array. This means that you can declare multiple files here. That can be useful if you want to reuse styles between just a couple of components.

That's the basic makeup of an Angular component. It's really just a supped up class file with specific definitions that your application will understand. With that information out of the way, let's generate a couple components in our example and explore some of the functionality that Angular components have.

Parent-Child Example

In my example, I am going to create a `ComponentInteractionModule`. I will add two components to the module, a `ParentComponent` and a `ChildComponent`. These names are not special to Angular, I am just using them for demonstration. I am going to create two mock children objects with data in my parent component. I will pass one child to one instance of the `ChildComponent` and the other child to a second instance. Both children will hold a collection (array) of futbols, that we will be passing back and forth between the children using the parent as a facilitator to demonstrate component IO. I'm going to build everything first and then I will discuss what I have done. Let's get started.

Setup the Demo: Create Module, Components, and Route

To setup, execute these commands:

- `ng g module core/component-interaction`
- `ng g component core/component-interaction/parent`
- `ng g component core/component-interaction/child`
- Open `core-routing.ts` and add a path to the `ParentComponent` called 'component-interaction'

```
1 {  
2   path: 'component-interaction',  
3   component: ParentComponent  
4 }
```

- Add the to the `SidebarComponent`

```
1 <div class="p-2">  
2   <a routerLink="/component-interaction">Component Interaction Example</a>  
3 </div>
```

With that you should be able to run `ng serve` in the example and be able to route to the using the sidebar.

Demo

As I stated above, in my `ParentComponent` I will add two child objects to hold some data. I will also add a `passBall(id:number)` function that will be used to "pass" a ball back and forth between the child objects. When I've done so, my `ParentComponent` looks like:


```

1 import { Component, OnInit } from '@angular/core';
2 @Component({
3   selector: 'app-parent',
4   templateUrl: './parent.component.html',
5   styleUrls: ['./parent.component.css']
6 })
7 export class ParentComponent implements OnInit {
8   child1: {futbols: Array<boolean>, name: string, id: number} = {
9     futbols: [true, true, true, true, true],
10    name: 'Child 1',
11    id: 1
12  };
13
14  child2: {futbols: Array<boolean>, name: string, id: number} = {
15    futbols: [true, true, true, true],
16    name: 'Child 2',
17    id: 2
18  };
19
20  constructor() { }
21
22  ngOnInit() {
23  }
24
25  passBall(id: number) {
26    if (id == 1) {
27      this.child1.futbols.pop();
28      this.child2.futbols.push(true);
29    } else if (id == 2) {
30      this.child2.futbols.pop();
31      this.child1.futbols.push(true);
32    }
33  }
34 }

```

Now I need to create my template for my ParentComponent. I am going to need to place my two instances of ChildComponent inside of this template. I'll add some classes to set the background colors of the content area that components are in.

```

1 <div class="parent">
2   <h6 class="text-center">I Am The Parent Component!</h6>
3   <div class="d-flex">
4     <div class="col-6 child-1">
5       <app-child [child]="child1" (emitPass)="passBall($event)"></app-child>
6     </div>
7     <div class="col-6 child-2">
8       <app-child [child]="child2" (emitPass)="passBall($event)"></app-child>
9     </div>
10  </div>
11 </div>

```

Before we setup our ChildComponent, let's talk about what we have done in the ParentComponent. In the template we just set up, there are two pieces of non-standard HTML where we have placed our ChildComponent inside of our ParentComponent, like so:

```

1 <app-child [child]="child1" (emitPass)="passBall($event)"></app-child>

```

Here we can see that we are using the selector "app-child" of ChildComponent to place it inside the ParentComponent.

When we add [child]="child1" to the selector, we are passing "child1" from the definition we declared in the ParentComponent as data for an input variable named "child".

When we add `(emitPass)="passBall($event)"`, we are telling our ParentComponent to detect an event from the ChildComponent with the name "emitPass". When the event occurs, it will fire the `passBall(id:number)` function we wrote, with the `$event` being the passed variable.

Now we just need to set up our ChildComponent with that input variable, and output event. Once we have done so, the `child.component.ts` will look like this:

```
1 import { Component, OnInit, Input, EventEmitter, Output } from '@angular/core';
2
3 @Component({
4   selector: 'app-child',
5   templateUrl: './child.component.html',
6   styleUrls: ['./child.component.css']
7 })
8 export class ChildComponent implements OnInit {
9   @Input() child: {futbols: Array<boolean>, name: string, id: number};
10  @Output() emitPass: EventEmitter<number> = new EventEmitter<number>();
11  constructor() { }
12
13  ngOnInit() {
14  }
15
16  passBall(){
17    this.emitPass.emit(this.child.id);
18  }
19 }
```

`@Input()` followed by the name of the input variable and type definition is how we have declared the child input.

`@Output()emitPass: EventEmitter<number> = new EventEmitter<number>()` declares our output variable `emitPass` as an `EventEmitter`, which is a built-in Angular class used to emit custom Events. Here we declared the type of payload of the event as a number. Declaring types throughout your application will help you build strong, testable code and improve your compilation.

Finally, our `passBall()` function will be used to fire our `emitPass` event, sending the id of our child input to the parent. Remember, the parent uses the id to know who to simulate the pass from/to.

Now we just add our template for our ChildComponent, we will need to display the futbols the child currently has and add a way to fire the `passBall()` function. Our template is going to look like this:

```

1 <div class="card">
2   <div class="card-body">
3     <div class="card-title">
4       {{child.name}}
5     </div>
6     <p class="card-text">
7       <span (click)="passBall()">
8         <i *ngFor="let ball of child.futbols" class="fa fa-futbol-o"></i>
9       </span>
10    </p>
11    <p>
12      Click a ball to pass it!
13    </p>
14  </div>
15 </div>

```

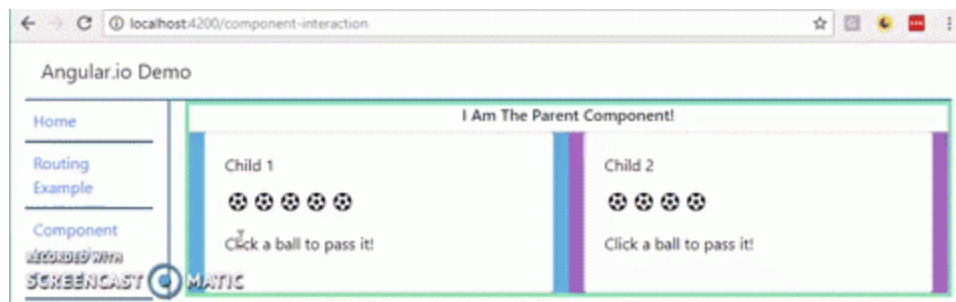
The interesting Angular bits:

{{child.name}} binds the variable with the template to display the name.

(click)="passBall()" is a built-in event that will fire our passBall() function when that span element is clicked.

*ngFor="let of child.futbols" is an Angular directive that instantiates an instance of the template within the context of the loop for every item in an iterable. In our case, we are just creating a font awesome futbol icon for every futbol in our futbols array. It's basically a for loop for elements.

Now, this example will look something like this (with styles added).



Conclusion

Hopefully that provides you with enough information to start adding different Angular components to your application. We have gone over what the different pieces of an Angular component are, looking at the different files, and how they are used together to make up an Angular component. We explored basics of inputs and outputs of components, including using an event emitter to send signals to a parent component to do something.

Chapter 5: Angular Component Lifecycle

In this chapter I will discuss Angular component lifecycles and how their lifecycle events can be leveraged. I will first give an overview of an Angular component's lifecycle and lifecycle events. I will then add a new example module to demonstrate when a component's lifecycle events are fired.

Angular Component Lifecycle

Every Angular component has a lifecycle. Actually, every Angular component and Angular directive have a lifecycle and the following information can be applied to both. The lifecycle is managed internally by Angular. So what is the lifecycle? According to Angular's documentation:

Angular creates it, renders it, creates and renders it's children, checks it when it's data-bound properties change, and destroys it before removing it from the DOM.

<https://angular.io/guide/lifecycle-hooks>

That is a very simple description of the sequence of events that an Angular component's life experiences. These events are called "Lifecycle Hooks". Developers can use these lifecycle hooks to do something (run some code) whenever one of these events occur. There are eight different lifecycle hooks that a developer can tap into in any component or directive. To do so, a developer just needs to add one of the eight function calls that correspond to the lifecycle event. Add the hooks you need and leave out the ones you don't.

8 Lifecycle Hooks

- `ngOnChanges()`
 - Used in pretty much any component that has an input.
 - Called whenever an input value changes
 - Is called the first time before `ngOnInit`

- `ngOnInit()`
 - Used to initialize data in a component.
 - Called after input values are set when a component is initialized.
 - Added to every component by default by the Angular CLI.
 - Called only once
- `ngDoCheck()`
 - Called during all change detection runs.
 - A run through the view by Angular to update/detect changes
- `ngAfterContentInit()`
 - Called only once after first `ngDoCheck()`
 - Called after the first run through of initializing
- `ngAfterContentChecked()`
 - Called after every `ngDoCheck()`
 - Waits till after `ngAfterContentInit()` on first run through
- `ngAfterViewInit()`
 - Called after Angular initializes component and child component content.
 - Called only once after is initialized
- `ngAfterViewChecked()`
 - Called after all the content is initialized and checked. (Component and child components).
 - First call is after `ngAfterViewInit()`
 - Called after every `ngAfterContentChecked()` call is completed
- `ngOnDestroy()`
 - Used to clean up any necessary code when a component is removed from the DOM.
 - Fairly often used to unsubscribe from things like services.
 - Called only once just before component is removed from the DOM.

In my experience as an Angular developer, I primarily use only four of these hooks. Mostly because I don't want to do something to a component after the content has already been checked.

- `ngOnChanges()`
- `ngOnInit()`
- `ngAfterViewInit()`
- `ngOnDestroy()`

The first two I use fairly frequently. They are very useful when dealing with input values or setting your component state based on outside data. The other two are very use case specific. If for some reason you need to do something after your component content has been set, use `ngAfterViewInit`. As I stated above, clean up your component with `ngOnDestroy()`. I am going to set up an example module to demonstrate the different hooks.

Example

I am going to add a new module to my `ng-example` repo. It's going to be called `lifecycle-hooks` and it has a parent component (`lifecycle-hooks`) and a child component for a `ngOnChanges` example called `changes-example`. I added routing to the sidebar to route to the `lifecycle-hooks` component and place the `changes-example` component inside of that component. Basically it's a parent component with a child component. Next, I am going to add all of the above lifecycle hooks to both components and `console.log()` the name of the hook that is called. I added "child" to the child's console log statements to differentiate the two. This is what the parent component looks like:

```
1 export class LifecycleHooksComponent implements OnInit {
2   constructor() { }
3   ngOnInit() {
4     console.log('ngOnInit');
5   }
6   ngOnChanges(){
7     console.log('ngOnChanges');
8   }
9   ngDoCheck(){
10    console.log('ngDoCheck');
11  }
12  ngAfterContentInit(){
13    console.log('ngAfterContentInit');
14  }
15  ngAfterContentChecked(){
16    console.log('ngAfterContentChecked')
17  }
18  ngAfterViewInit(){
19    console.log('ngAfterViewInit');
20  }
21  ngAfterViewChecked(){
22    console.log('ngAfterViewChecked');
23  }
24  ngOnDestroy(){
25    console.log('ngOnDestory');
26  }
27 }
```

Again, the child is identical but with "child" as a part of the console.log statement. So now, if we run ng serve we will see the order that these are fired during initialization.

```
ngOnInit          lifecycle-hooks.component.ts:17
ngDoCheck         lifecycle-hooks.component.ts:25
ngAfterContentInit lifecycle-hooks.component.ts:29
ngAfterContentChecked lifecycle-hooks.component.ts:33
child ngOnInit    changes-example.component.ts:14
child ngDoCheck   changes-example.component.ts:25
child ngAfterContentInit changes-example.component.ts:29
child ngAfterContentChecked changes-example.component.ts:33
child ngAfterViewInit changes-example.component.ts:37
child ngAfterViewChecked changes-example.component.ts:41
ngAfterViewInit   lifecycle-hooks.component.ts:37
ngAfterViewChecked lifecycle-hooks.component.ts:41
```

This should give you a decent picture of the order of operations for the lifecycle hooks on initialization. The component initializes, the content is initialized and is checked, then the child runs through its initialization and checks. Finally, the component view (including children) declares it has been initialized and has been checked. I just want to add a simple event to the parent component that manipulates some input value to the child. All I am going to do is add a button in the parent that increases a number by one when it is clicked. That number is going to be passed as an input to the child and displayed there. The additions to the parent .ts look like this.

```
1 export class LifecycleHooksComponent implements OnInit {
2
3   num:number = 0;
4
5   constructor() { }
6
7   add(){
8     console.log('CLICKED')
9     this.num++;
10  }
11
12  ...
13 }
```

The parent looks like this. (bootstrap classes added to make it look presentable).

```
1 <div class="w-100 pt-4 justify-content-center d-flex">
2   <button class="btn" (click)="add()"></button>
3   <div class="pl-4">
4     <app-changes-example [num]="num"></app-changes-example>
5   </div>
6 </div>
```

And then I am just displaying the input "num" in the child .

```
1 <h4>{{num}}</h4>
```


So now when we click the '+' button, an event is fired and the components will experience a series of lifecycle events. Let's see what the console looks like on initialization.

```
ngOnInit lifecycle-hooks.component.ts:17
ngDoCheck lifecycle-hooks.component.ts:25
ngAfterContentInit lifecycle-hooks.component.ts:29
ngAfterContentChecked lifecycle-hooks.component.ts:33
child ngOnChanges ← lifecycle-hooks.component.ts:21
child ngOnInit changes-example.component.ts:14
child ngDoCheck changes-example.component.ts:25
child ngAfterContentInit changes-example.component.ts:29
child ngAfterContentChecked changes-example.component.ts:33
child ngAfterViewInit changes-example.component.ts:37
child ngAfterViewChecked changes-example.component.ts:41
ngAfterViewInit lifecycle-hooks.component.ts:37
ngAfterViewChecked lifecycle-hooks.component.ts:41
```

Notice now that the ngOnChanges event fires in the child. That's because the child now has an input value that ngOnChanges can detect. Now I am going to click the button and fire the event. The console will show the hooks that fire when the event occurs.

```
CLICKED lifecycle-hooks.component.ts:12
ngDoCheck lifecycle-hooks.component.ts:25
ngAfterContentChecked lifecycle-hooks.component.ts:33
child ngOnChanges changes-example.component.ts:21
child ngDoCheck changes-example.component.ts:25
child ngAfterContentChecked changes-example.component.ts:33
child ngAfterViewChecked changes-example.component.ts:41
ngAfterViewChecked lifecycle-hooks.component.ts:41
```

Here we can see the parent runs a check of itself, once that components direct content is checked, the child component does it's own check. However, the child's check runs after changes are detected and handled.

Bonus:

SimpleChanges is a class that can be hooked into ngOnChanges that will give you some context of the changes occurring to your input values. Every input value will have a few properties when they change:

- previousValue - The value before the change
- currentValue - The new value after the change
- firstChange - A boolean value set true it is the first change for the input value.

Its use looks something like this.

```
1 ngOnChanges(changes: SimpleChanges){
2   if(changes.num.currentValue >> changes.num.previousValue){
3     console.log('num went up from: ' + changes.num.previousValue + ' to ' + changes.num.currentValue);
4   }
5 }
```

Here “num” is the name of the input that is experiencing a change.

Conclusion

With I have provided you an overview of what lifecycle hooks are, the order in which lifecycle events occur and a way to leverage those events. Remember, these events occur in both components and directives. Be sure to choose carefully which event is needed for what you are doing. In my next chapter, we will explore template driven Angular forms.

Chapter 6: Template Driven Angular Forms

In this we will explore Angular forms built using template syntax. Angular provides developers two ways to build forms, one uses template syntax and the other is a model-driven approach. Forms built using the approach are called “Reactive Forms” and I will follow this chapter up with a section on Reactive Forms.

To build a template driven Angular form you should be familiar with Angular’s [template syntax](#). Using the template syntax and a few form-specific directives will allow developers to build robust forms using primarily code.

Module Setup

For this example, we are going to create a FormsExampleModule that will hold all of our necessary content for the example. Generating this module and the following components. I am going to add some setup for the reactive chapter here as well but feel free to skip the reactive component if you are not interested. The “core” portion of the following commands reflects how this example repo has been if you aren’t following along from previous examples or your project is differently, then “core” can be omitted.

1. `ng g module core/forms-example`
2. `ng g component core/forms-example`
3. `ng g component core/forms-example/template-example`
4. `ng g component core/forms-example/reactive-example`

I updated the routing inside the example application as I have previously for other examples. I won’t demonstrate those steps again here.

Our generated FormsExampleModule is a bit like a application at this point. We are going to have to provide the module with the correct application pieces to build our template driven Angular form. What I mean by that; we are going to be using form specific directives to build our Angular form, thus we need to bring those directives into our module to use them. We do this by importing the “FormsModule” and providing it to the “imports” array of our FormsExampleModule. After we can get started building our template driven Angular form.

User Class

we are going to create our "User" class. This will act as our object model that will hold the information for our user. Create a user.ts file somewhere in your project and then create the following user class.

```
1 export class User {
2   firstName: string;
3   lastName: string;
4   role: string;
5   notes?: string;
6 }
```

Pretty standard stuff for a simple user class that we can use for our data binding. let's build out the form.

Building Our Template

Now we are going to use standard to build out the Angular form template. I have placed the template example component inside of a bootstrap card in the forms-example.component.html.

```
1 <div class="w-50 p-2">
2   <div class="card">
3     <div class="card-header text-center">
4       <h5>Template Form Example</h5>
5     </div>
6     <div class="card-body">
7       <app-template-example></app-template-example>
8     </div>
9   </div>
10 </div>
```

Add the following code block to the template-example.component.html file. The html for the form itself looks like this:

```
1 <form>
2   <div class="form-group">
3     <label for="firstName">First Name</label>
4     <input class="form-control" name="firstName" id="firstName" type="text">
5   </div>
6   <div class="form-group">
7     <label for="lastName">Last Name</label>
8     <input class="form-control" name="lastName" id="lastName" type="text">
9   </div>
10  <div class="form-group">
11    <label for="role">User Role</label>
12    <select class="form-control" name="role" id="role">
13      <option>Guest</option>
14      <option>Admin</option>
15      <option>Owner</option>
16      <option>Operator</option>
17    </select>
18  </div>
19  <div class="form-group">
20    <label for="notes">Additional Notes</label>
21    <textarea class="form-control" name="notes" id="notes" rows="6" placeholder="Add additional notes">
22  </div>
23 </form>
```

This is all standard with some additional bootstrap classes for formatting. Now we have a form built, but it can't really do anything at the moment. Now we have to use our form directives that we imported with our FormsModule earlier.

Initializing Our User

Next in the we need to initialize our user that we are going to be binding to the form. Import the class we created previously and initialize how you see fit. I am going to have it hold some mock data for demonstration purposes.

```
1 user: User = {
2   firstName: 'New',
3   lastName: 'User',
4   role: 'Guest',
5   notes: undefined
6 };
```

ngForm and ngModel

ngForm and ngModel are the Angular form directives that we are going to add to our current form to provide Angular with the right information for the framework to internally build of the form. Internally, Angular is going to create a "FormGroup" object. Somewhat interesting, a FormGroup is what you would build yourself if you were building a Reactive Form. So deep down, Angular treats Template Driven Forms and Reactive Driven Forms in the same way. The difference being how you build them and the functionality that comes with each direction. I will demonstrate how to view your Template Driven Form Group in a moment. First, let's use the ngForm directive on our form so Angular will know to make the FormGroup out of the form.

```
1 <form #userForm="ngForm">
2   <!--form content-->
3 </form>
```

Once we do this, we can actually use @ViewChild('userForm') to grab the instance of the Angular form in our component file. Now there isn't any real practical reason to do this in a real development environment but by logging this element we can get a good look at what Angular creates in terms of the FormGroup, which I found to be some valuable insight.

Add this code snippet to the template-example.component.ts

```
1 @ViewChild('userForm') userForm: ElementRef;
2 logForm(){
3   console.log(this.userForm);
4 }
```

And this to template-example.component.html

```
1 <div class="d-flex w-100">
2   <button class="btn pull-right btn-primary" (click)="logForm()">Log Form</button>
3 </div>
```

This will allow us to view the state of our form through Angular's eyes at any time. Now as our form is setup, if we run the application and log the form, we will get this.

```
template-example.component.ts:33
▼ NgForm {submitted: false, _directives: Array(0), ngSubmit: EventEmitter, form: FormGroup}
  ▶ control: FormGroup
  ▼ controls: Object
    ▶ __proto__: Object
    dirty: false
    disabled: false
    enabled: true
    errors: null
    ▶ form: FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f, pristine: true, touched: false, -}
    ▶ formDirective: NgForm
      invalid: false
    ▶ ngSubmit: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped: false, hasError: false, -}
    ▶ path: Array(0)
      pending: false
      pristine: true
      status: "VALID"
    ▶ statusChanges: EventEmitter
      submitted: false
      touched: false
      untouched: true
      valid: true
    ▶ value: Object
    ▶ valueChanges: EventEmitter
    ▶ _directives: []
    ▶ __proto__: ControlContainer
```

This is the state of the form before it is touched at all (you can see this because pristine = true). As you can see, there is some interesting information here. Remember we really haven't done anything but build an html form and add the ngForm directive. Now I highlighted the controls object to show that it is currently empty. As we add our data binding we will see this object fill out. Let's bind our first name input to our user and see what the form looks like after that.

To add a template driven control to a form you need two things:

1. a unique name for the control (we already have added a name to each of our controls above)
2. `[(ngModel)]` is what is used to bind the pieces of our user object to the form control

Additionally, I recommend adding directive. It's going to be useful for error messaging later, it basically provides a way to access the form control in other places within the html. After adding all of that, our firstName input field is going to look like this:

```
1 <input class="form-control" name="firstName" id="firstName" [(ngModel)]="user.firstName" #firstName="ng
```

Again,

-[(ngModel)]="user.firstName" binds the data model to the input field.

-name="firstName" a unique name for Angular to use to build the FormGroup.

-#firstName="ngModel" adds the ngModel directive so we can use "firstName" to access the control object for "firstName" later on.

Now when we run the program and log our form, we will see the first name form control object in the controls object of our FormGroup.

```
template-example.component.ts:33
▼ NgForm {submitted: false, _directives: Array(1), ngSubmit: EventEmitter, form: FormGroup}
  control: {...}
  controls: Object
    ▼ firstName: FormControl {validator: null, asyncValidator: null, _onCollectionChange: f, pristine: true, touched: false, ...}
      __proto__: Object
      dirty: {...}
      disabled: {...}
      enabled: {...}
      errors: {...}
      form: FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f, pristine: true, touched: false, ...}
      formDirective: {...}
      invalid: {...}
      ngSubmit: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped: false, hasError: false, ...}
      path: {...}
```

Validation

I just want to provide a simple example of validation for your input fields. In my experience, it's more difficult to do validation for a template driven form than a reactive form. The same validation is possible for both, I just find the validators for reactive forms to be easier to use. However, we can require fields, give them a minimum length and add forbidden values with template driven Angular forms. If that is all the validation you are really concerned about then a template driven form is definitely a viable option. Below is an example of required validation on the first name input field:

```
1 <input class="form-control" name="firstName" id="firstName" [(ngModel)]="user.firstName" #firstName="r
2   <div *ngIf="!firstName.valid && !firstName.pristine" class="alert alert-danger">
3     <span *ngIf="firstName.errors.required">
4       First Name is required
5     </span>
6   </div>
```

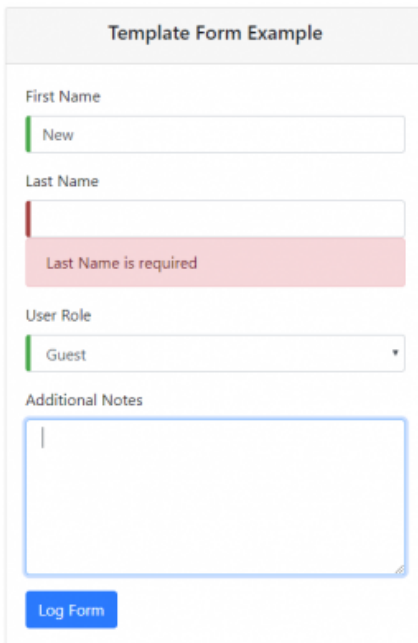
What I have done is added "required" to the input field, which Angular picks up on and adds to the form control object. I am checking the form control object's properties of "valid" and "pristine" to see if there are any errors to display. If there are errors, then I check the form control "errors" object to see if it is the "required" field that is missing. If there was additional validation (most [native HTML form validation](#) will work), then there might be additional errors in the error object that we would want to check for different messaging.

Bonus: Styling

Along with Angular creating a FormGroup object for you, it also applies classes to the input and form elements based on valid/invalid properties. “.ng-valid” and “.ng-invalid” classes will be applied as valid/invalid states change. To demonstrate this you can add the following styling class definitions to template-example.component.css

```
1 .ng-valid[required], .ng-valid.required {
2   border-left: 5px solid #42A948; /* green */
3 }
4 .ng-invalid:not(form) {
5   border-left: 5px solid #a94442; /* red */
6 }
```

This will add a border on the left of input fields that are required based on their state. Once our form is completed with validation, data binding, and styling it will look something like this.



The screenshot shows a form titled "Template Form Example" with the following fields and states:

- First Name:** Input field containing "New" with a green border on the left.
- Last Name:** Empty input field with a red border on the left and a red error message "Last Name is required" below it.
- User Role:** Dropdown menu with "Guest" selected and a green border on the left.
- Additional Notes:** Textarea field with a blue border.
- Log Form:** A blue button at the bottom.

Conclusion

Above I have demonstrated how to build a template driven Angular form with validation and data binding. For a working example of this, check out [my example repo](#) under "Forms Example". I recommend digging through the "FormGroup" object that is created by Angular, there may be some useful pieces of information you want to use in there.

Chapter 7: Angular Reactive Forms Tutorial

In this chapter, we will explore forms using Angular's FormBuilder, FormGroup, and FormControl classes. Angular provides developers two ways to build forms, one uses template syntax (which we explored in the previous chapter) and the other is this model-driven approach. Choosing which form building technique to use is completely up to the developer. Angular reactive forms are usually easier to test since you are working with objects and models. In my experience, if you are going to need to use a robust set of validation, then you should build Angular reactive forms.

Creating an Angular Reactive Form with FormGroup, FormControl, and FormBuilder

In Angular, a reactive form is a [FormGroup](#) that is made up of [FormControls](#). The [FormBuilder](#) is the class that is used to create both FormGroups and FormControls. I encourage you to check out those links to see the full class definitions of all three. Below I am going to demonstrate how to build a FormGroup using FormBuilder for our user form.

Inside of the reactive-example.component.ts, we will add the following. We will start with a basic form and add on to it as we go.

```
1 export class ReactiveExampleComponent implements OnInit {
2   userForm: FormGroup;
3   roles: Array<string> = [
4     'Guest',
5     'Admin',
6     'Owner',
7     'Operator'
8   ];
9   constructor(private formBuilder: FormBuilder) {
10    this.userForm = this.formBuilder.group({
11      'firstName': [''],
12      'lastName': [''],
13      'role': [''],
14      'notes': ['']
15    });
16  }
17  ngOnInit() {
18  }
19 }
```


You will need to import FormBuilder and FormGroup from @angular/forms. What we now have done is created a userForm of type FormGroup with 4 FormControl using the FormBuilder. The roles array will be used in our template. These controls are pretty bare bones at the moment but we will enhance these down the line. let's build out our template.

Binding to a Template with formControlName

Our template is going to look very similar to what we built in the template driven forms example. However, instead of using ngModel and various form directives, we will bind our userForm to our template with [FormGroup] and formControlName. Add the following html code to the reactive-forms-example.component.html:

```
<form [formGroup]="userForm">
  <div class="form-group">
    <label for="firstName">First Name</label>
    <input class="form-control" name="firstName" id="firstName" type="text" formControlName="firstName">
  </div>
  <div class="form-group">
    <label for="lastName">Last Name</label>
    <input class="form-control" name="lastName" id="lastName" type="text" formControlName="lastName">
  </div>
  <div class="form-group">
    <label for="role">User Role</label>
    <select class="form-control" name="role" id="role" formControlName="role">
      <option *ngFor="let userRole of roles" [ngValue]="userRole">{{userRole}}</option>
    </select>
  </div>
  <div class="form-group">
    <label for="notes">Additional Notes</label>
    <textarea class="form-control" name="notes" id="notes" formControlName="notes" rows="6"
      placeholder="Add additional notes here."></textarea>
  </div>
</form>
```

So what we have done is provided the <form> an instance of a FormGroup (the userForm we initialized in the constructor). After doing that, we created input fields for each FormControl in the userForm and we bound the input fields to the userForm using formControlName, setting the formControlName to the name of our form control. Simple enough right? Now we have a form template built with data bound to our ReactiveExampleComponent.

To see the state of our form I am going to add a button that just calls a function in our component that console.logs(this.userForm).

```
reactive-example.component.ts:38
FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f, pristine: true, touched: false, ...}
  asyncValidator: null
  controls:
    ▶ firstName: FormControl {validator: null, asyncValidator: null, _onCollectionChange: f, ...}
    ▶ lastName: FormControl {validator: null, asyncValidator: null, _onCollectionChange: f, ...}
    ▶ notes: FormControl {validator: null, asyncValidator: null, _onCollectionChange: f, ...}
    ▶ role: FormControl {validator: null, asyncValidator: null, _onCollectionChange: f, ...}
    ▶ __proto__: Object
  dirty: false
  disabled: false
  enabled: true
  errors: null
  invalid: false
  parent: undefined
  pending: false
  pristine: true
  ▶ root: FormGroup
  status: "VALID"
  ▶ statusChanges: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped: false}
  touched: false
  untouched: true
  updateOn: "change"
  valid: true
  validator: null
  ▶ value: {firstName: "", lastName: "", role: "", notes: ""}
  ▶ valueChanges: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped: false}
  ▶ _onCollectionChange: f ()
  ▶ _onDisabledChange: []
  ▶ __proto__: AbstractControl
```

This is what we get when the form is logged after initialization and before it is touched. Logging your form is a good way to see what Angular has for the state of your form and can be valuable for debugging. As you dig in, you can see the state of the form controls and their values.

Initializing Form Data and Validation

Next, we will add some simple validation to the form, as well as demonstrate how to initialize the form with some data. Use the User class to initialize a user with default data. After that we use a Validators class to set validation properties for each control. This is an optional section and can remain blank. You'll notice that I have used an array here, that allows for multiple Validators to be used on a control. There are some pretty handy [validators](#) that are built in by Angular. Custom validators can be written but that is outside the scope of this post.

```

1  user: User = {
2    firstName: 'New',
3    lastName: 'User',
4    role: 'Guest',
5    notes: undefined
6  };
7  constructor(private formBuilder: FormBuilder) {
8    this.userForm = this.formBuilder.group({
9      'firstName': [this.user.firstName, [Validators.required]],
10     'lastName': [this.user.lastName, [Validators.required]],
11     'role': [this.user.role, [Validators.required]],
12     'notes': [this.user.notes, [Validators.maxLength(45)]]
13   });
14 }

```

It should be noted that there are a ways to do this, and this is just one of them. You could choose to create individual controls, but that would be the functionality of the FormGroup. The nice thing about the FormGroup is that it allows the developer to easily check if the whole form is valid. As you see in the console.log(this.userForm), the form has an invalid property that is a boolean.

NgSubmit

Finally, I just want to demonstrate how to use ngSubmit with your Angular reactive forms. It's pretty straight forward, you tag the form with a function you want to be called when the form is submitted. Then you just add a button of type somewhere in the form, usually the bottom. Looks something like this:

```

1 <form [formGroup]="userForm" (ngSubmit)="logFormValue()">
2   ....
3   <button type="submit" class="btn btn-default" [disabled]="!userForm.valid">Submit</button>
4 </form>

```

Using the valid property of the userForm we are able to prevent submitting an invalid form. Once valid and submitted, the logFormValue() function will be called. This same functionality can be used in template driven forms.

Conclusion

In this chapter, I have demonstrated how to build an Angular Reactive Form, with some basic validation and data binding. I would encourage you to familiarize yourself with the FormGroup, FormControl, and FormBuilder classes. There is a bunch of functionality in those classes that will make building robust forms easier. I have added some additional functionality to [my example repo](#). I encourage you to check it out and use it as a reference for your own projects.

Chapter 8: Getting started with Electron and Angular CLI

In this Electron tutorial, I will walk through the steps you can take to turn an Angular application into a native desktop application using Electron. A technology stack using Angular and Electron allows developers to build desktop applications for Windows, Mac, and Linux using web technologies.

This Electron tutorial will focus mainly on Electron and how to get started developing an Electron application with Angular. If you are unfamiliar with what Electron is, it essentially is a chrome browser dedicated to running a single application. Electron also has to natively interact with a computer's operating system. Electron is packaged with your application, meaning, there will be consistency across operating systems in the same way a chrome browser is consistent across operating systems. Thus, you can write a single web application, with a single code base and use that application with Electron to turn a web application into desktop applications.

I've also created a video to accompany this tutorial that walks through the different sections of this blog post. You can view that [here](#) as well as get more description below.

Electron has a core set of API's it uses to interact with the user's operating system. On top of that, it includes all of Chromium's APIs, all of NodeJS's modules and can support modules.

Electron Forge

Electron has a fairly large open source community, with that comes a wide variety of options on how you want to leverage the framework. Thus, setting up an Electron application can be done in a variety of ways. One way, the way I am going to demonstrate in this Electron tutorial, is using Electron Forge:

Electron Forge is Electron's command line interface (CLI) tool. By leveraging the Electron Forge CLI in the way I am about to demonstrate, we are able to add one file and modify the package.json in an Angular CLI generated project to create a desktop application. Let's get started!

Create Your Main Script

Electron uses the main script defined in your package.json to start up. Meaning it will check your package.json for the "main" script and execute whatever it finds. So, we will need to create this script in our project. The main script is going to start the electron application, create a window, and the load a specified URL. The URL can be a remote web URL if you feel like creating a desktop app out of your favorite website (this does have some limitations). In our case, we are going to have our main script load the entry point to our application.

Inside of your Angular CLI generated project, add the following code block to a file that I am going to call electronMain.js. This file comes from Electron's documentation for getting started <https://electron.atom.io/docs/tutorial/quick-start/> with a few changes to it to make it work with our Angular application. Create the file in the src/ folder of your project (Electron Forge won't allow it to be in the same directory as package.json).

```
1 const { app, BrowserWindow } = require('electron')
2 const path = require('path')
3 const url = require('url')
4 let win;
5 function isDev() {
6   return process.mainModule.filename.indexOf('app.asar') === -1;
7 }
8 function createWindow() {
9   win = new BrowserWindow({ width: 800, height: 600 })
10  win.loadURL(url.format({
11    pathname: path.join(__dirname, '../dist/index.html'),
12    protocol: 'file',
13    slashes: true
14  }));
15  if (isDev()) {
16    win.webContents.openDevTools()
17  }
18  win.on('closed', () => {
19    win = null
20  })
21 }
22 app.on('ready', createWindow)
23 app.on('window-all-closed', () => {
24   if (process.platform !== 'darwin') {
25     app.quit()
26   }
27 })
28 app.on('activate', () => {
29   if (win === null) {
30     createWindow()
31   }
32 })
```

**If you are using Angular v6+, your pathname will be: '../dist/appName/index.html'

I have made a few changes from what is found in the Electron Getting Started guide for this Electron tutorial. The changes I have done are for the Electron app to use a loadURL that points at our dist/ folder and loads the index.html file. If you are a beginner to Angular and the CLI, when you run "npm run build" in a CLI generated project, it builds the application and puts it into the dist/ folder with index.html as your entry point for the app. Additionally, I have added a check for if the application is being run in dev mode, the isDev() function. If it is in dev mode, the Electron app will open Chrome's dev tools when the application starts up, which I find very useful for development.

Next, we need to update our package.json with a couple things to get this all to work. An Electron app requires a few pieces of information in your package.json to work:

- name (should be generated by Angular CLI)
- version (should be generated by Angular CLI)
- main (set to the main script we just created)
- author
- description

Additionally, we are going to add a few packages to our dependencies, three scripts, and some config information to get Electron Forge to work.

```
1  "main": "src/electronMain.js",
2  "author": "rroot",
3  "description": "Example Angular and Electron"
4  "scripts": {
5    ...
6    "forge-start": "electron-forge start",
7    "package": "electron-forge package",
8    "make": "electron-forge make",
9    ...
10 },
11 ...
12 "config": {
13   "forge": {
14     "make_targets": {
15       "win32": [
16         "squirrel"
17       ],
18       "darwin": [
19         "zip"
20       ],
21       "linux": [
22         "deb",
23         "rpm"
24       ]
25     },
26     "electronPackagerConfig": {
27       "packageManager": "npm"
28     },
29     ...
30   }
31 }
```

```

29   "electronwinstallerConfig": {
30     "name": "ng-example"
31   },
32   "electronInstallerDebian": {},
33   "electronInstallerRedhat": {},
34   "github_repository": {
35     "owner": "",
36     "name": ""
37   },
38   "windowsStoreConfig": {
39     "packageName": "",
40     "name": "ngExample"
41   }
42 }
43 },
44 "dependencies": {
45   ...
46   "electron-compile": "^6.4.2",
47   "electron-squirrel-startup": "^1.0.0",
48   ...
49 },
50 "devDependencies": {
51   ...
52   "electron": "^2.0.0",
53   "electron-forge": "^5.2.0",
54   "electron-prebuilt-compile": "2.0.0",
55   ...
56 }

```

Electron Forge has a set of templates for React, Angular, Vue.JS, and ES6/7. To get the configuration information above, I just ran the init command with the Angular flag set for the template in an empty project folder. A barebones template for Electron and Angular was generated. Then I took that information and put it into my Angular CLI generated project.

One final step we will need to do is update the href used by the Angular application to use a relative path. This allows Electron to find the resources of the project. In all honesty, I'm not a big fan of this step, but it works and it also doesn't break the web version of your app. To do so, open index.html and modify the base to look like this: `<base href="./">` (Just adding a period before the slash to use a relative path to our project.)

With that we can build and run our electron app to verify we have set our project up correctly.

Building and Running the Application

Make sure you run an npm install after updating your package.json with the electron dependencies. Then you can build the application like any other Angular application, "npm run build". Now just run the start script: npm run start-electron, and an Electron window should open up with your Angular CLI application running inside.

If you want to check that your web application still works, run ng serve and navigate to localhost:4200 and you should find the same results.

Make and Package

This is my favorite part about using Electron Forge. With that simple setup, not only are we able to get started developing an Electron Application with Angular in a short amount of time, but we are able to make distributables of the desktop application as well. As a web developer with no desktop development experience, I am able to build and distribute a desktop application using the skills I already have. So awesome!! All you have to do is run the make command script we added to our package.json and Electron Forge will do the work for you from there. Give it a shot!

Two things to note about make:

1. Electron Forge is going to build the target corresponding to the OS you are running the command on. Windows makes windows, Mac makes mac... etc.
2. To properly distribute Mac builds, you will have to set up code signing. I am not going to cover doing that here but it's a pretty straight forward process.

Additional Steps for Improving Development Process

With the setup described above, you will have to run "npm run build" and "npm run electron" for every change that you make while developing. This is not ideal, so lets speed up this development process a bit.

Add a script to package.json that will rebuild the application whenever changes are made to the codebase. I choose to use ng build without the --prod flag because at this time, rebuilding without aot is significantly faster. If you are on Angular v6, aot is used by default and the rebuilds are super fast. Here's the script:

- "build-watch": "ng build --op dist -w"

You will need to use two terminals, in the first run "npm run build-watch" and in the second run "npm run electron-start". Use the second terminal to kill and restart the electron app. This will work well for most development, however I have run into times where build-watch doesn't catch some changes, like adding new modules. If you are running into errors you don't understand, kill build-watch and restart it. You shouldn't be in that situation often but it's good to know.

Summary

With that, you are setup to start building your desktop application using Angular and Electron.

Chapter 9: Using Bootstrap 4 with Angular

In this final chapter, I will discuss what's new with Bootstrap 4 and how it aligns well with Angular. Before I get started, I should mention that there is a Material library that is supported by the Angular team. It provides all sorts of components that play well with Angular, and if you are already familiar with material, I would encourage you to explore the @angular/material library. With that said, it doesn't hurt to explore your options. The following will provide a quick rundown of Bootstrap 4. I have also added some examples to my ng-example repo that you can find at <https://github.com/rmroot/ng-example>.

The Approach to Bootstrap 4

Per the Bootstrap documentation, found [here](#), the team has a set of guides that steer their approach:

1. Components should be responsive and mobile-first
2. Components should be built with a base class and extend via modifier classes
3. Component states should obey a common z-index scale
4. Whenever possible, prefer a HTML and CSS implementation over Javascript
5. Whenever possible, use utilities over custom styles
6. Whenever possible, avoid enforcing strict HTML requirements (child selectors)

Looking through this list, a couple of these items stand out as aligning closely with Angular's approach to styling. Specifically, number four. In an Angular app, all styling should be done with CSS and HTML. You shouldn't be accessing the Dom or be using JQuery to apply classes to your elements. Angular provides two powerful directives for styling your elements without the use of JavaScript, NgClass and NgStyle. With those two directives, you shouldn't need to access the Dom to manipulate the style of any element.

**It is possible to access Dom elements without JQuery or window calls with ElementRef but even Angular's documentation discourages its use for security reasons.

Now I will admit that number five may seem to be a bit counter intuitive to an Angular since every component has it's own custom style sheet. But if done properly, those style sheets can be used for modifier classes (number two), and your application styling can still maintain the Bootstrap approach. If you do find the need to create custom styles, I would encourage you to use a BEM (Block Element Modifier) methodology, because it closely aligns with Bootstrap's approach. For more information on BEM, visit <http://getbem.com/introduction/>

Finally, numbers 1, 3, and 6 are just going to make your life easier. I'm a firm believer in a class first approach when styling web applications, it provides much more flexibility in your HTML.

Adding Bootstrap 4

Adding Bootstrap to your Angular application is an easy process. Just install it and import it to the global style sheet provided by the Angular CLI.

1. `npm install bootstrap@latest --save-dev`
2. Open `styles.css` and add the following
 - `@import "~bootstrap/dist/css/bootstrap.css";`

That's it. Now you have the ability to use all of Bootstrap's styles and utilities.

What Stands Out in Bootstrap 4

My two personal favorite pieces of Bootstrap 4 are the Card component and the addition of Flexbox utilities. Cards can be used as content containers with all sorts of variety and options that can be applied to them. Using cards is an easy way to group and cleanly layout a set of content in your application. They also work seamlessly with the grid system, so using multiple cards for repetitive content (with `*ngFor`) is a breeze. In addition to the typical Bootstrap grid system of rows and columns, the added Flexbox utilities also are a huge help in getting your content exactly where you want it. It gives you the full power of Flexbox CSS in the form of classes.

Summary

I use Bootstrap 4 and Angular on a daily basis for my current project, and I am constantly finding new ways to leverage it to make my life easier. It gives you a large set of utility type classes that can be applied as-is or overwritten in a given component. Sometimes a developer just wants a way to center content on a web page without having to write their own CSS, and that's the sort of thing Bootstrap 4 provides for you.