

Beginning Angular 2 with TypeScript



Greg Lim

Beginning Angular 2 with Typescript

Greg Lim

Copyright © 2017 Greg Lim

All rights reserved.

COPYRIGHT © 2017 BY GREG LIM

ALL RIGHTS RESERVED.

NO PART OF THIS BOOK MAY BE REPRODUCED IN ANY FORM OR BY ANY ELECTRONIC OR MECHANICAL MEANS INCLUDING INFORMATION STORAGE AND RETRIEVAL SYSTEMS, WITHOUT PERMISSION IN WRITING FROM THE AUTHOR. THE ONLY EXCEPTION IS BY A REVIEWER, WHO MAY QUOTE SHORT EXCERPTS IN A REVIEW.

FIRST EDITION: FEBRUARY 2017

Table of Contents

PREFACE

CHAPTER 1: INTRODUCTION

CHAPTER 2: ANGULAR 2 QUICKSTART

CHAPTER 3: RENDERING DATA AND HANDLING EVENTS

CHAPTER 4: BUILDING RE-USABLE COMPONENTS

CHAPTER 5: CONTROLLING RENDERING OF HTML

CHAPTER 6: TEMPLATE DRIVEN FORMS

CHAPTER 7: MODEL DRIVEN FORMS

CHAPTER 8: INTRODUCTION TO OBSERVABLES

CHAPTER 9: CONNECTING TO SERVER

CHAPTER 10: BUILDING SINGLE PAGE APPS WITH ROUTING

CHAPTER 11: STRUCTURING LARGE APPS WITH MODULES

CHAPTER 12: C.R.U.D. WITH FIREBASE

ABOUT THE AUTHOR

PREFACE

About this book

Angular 2 is one of the leading frameworks to develop apps across all platforms. Reuse your code and build fast and high performing apps for any platform be it web, mobile web, native mobile and native desktop. You use small manageable components to build a large powerful app. No more wasting time hunting for DOM nodes!

In this book, we take you on a fun, hands-on and pragmatic journey to master Angular 2 from a web development point of view. You'll start building Angular 2 apps within minutes. Every section is written in a bite-sized manner and straight to the point as I don't want to waste your time (and most certainly mine) on the content you don't need. In the end, you will have what it takes to develop a real-life app.

Requirements

Basic familiarity with HTML, CSS, Javascript and object-oriented programming

Contact and Code Examples

Please address comments and questions concerning this book to support@i-ducate.com.

Code examples can be also be obtained by contacting me at the same.

CHAPTER 1: INTRODUCTION

1.1 What is Angular 2?

Angular is the leading framework for building Javascript heavy applications. It is often used to build ‘Single Page Apps’ or SPA for short. What is a single page app? In a standard web application, when we click on a link, the entire page is reloaded. In a SPA, instead of reloading the entire page, we reload only the view whose content is requested. A SPA also keeps track of history, so if a user navigates using back and forward buttons, the application is reasserted in the right state. All these provide a fast and fluid experience for the user. Gmail is a good example of a SPA.

There are other frameworks out there that provide similar functions, so why Angular? Angular is one of the leading frameworks in this space. It has been around for quite a few years, it has a huge community support, it is backed by Google, and demand for Angular developers are constantly increasing.

In this book, I will teach you about Angular 2 from scratch in a step by step fashion. It doesn’t matter whether you are familiar with Angular 1 or not because Angular 2 is an entirely new framework. I will not be touching on Angular 1 and how it is different from Angular 2 because not every reader of this book is familiar with Angular 1 and we do not want to distract you with the old way of development. If you have an existing Angular 1 application that you want to upgrade to Angular 2, your best source is the Angular website. They have documented processes and strategies on how to upgrade to Angular 2. You can run Angular 1 and 2 side by side in the Angular website, and progressively upgrade one module at a time.

We will be using Typescript for Angular 2 development. Why not Javascript? Typescript is actually a superset of Javascript (fig. 1.1.1).

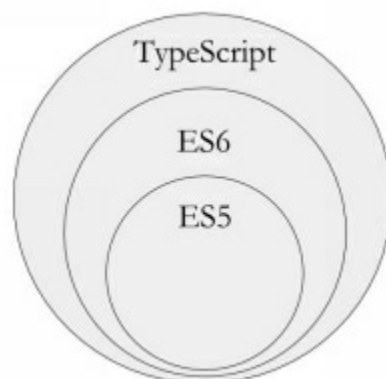


fig. 1.1.1

Any valid Javascript code is valid Typescript which means that you do not have to learn a new programming language. Typescript brings some useful features that are missing in the current version of Javascript supported by most browsers. It has modules, classes, interfaces, access modifiers like private and public, intellisense and compile time checking, so we can catch many programming errors during compile time.

In the course of this book, you will build an application where you can input search terms and receive the search results via Spotify RESTful api (fig. 1.1.2).

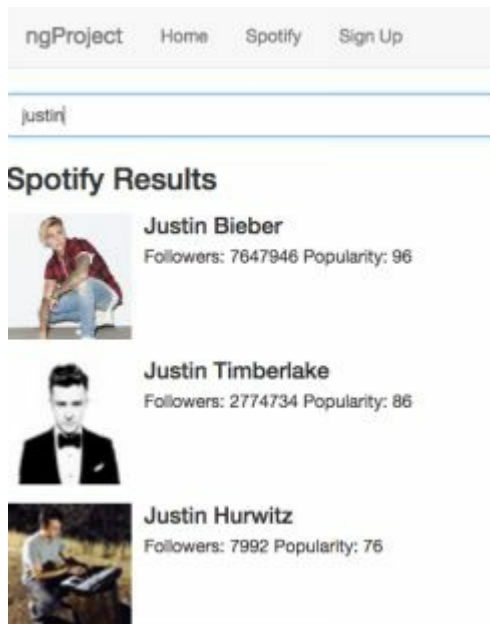


figure 1.1.2

At the end, you will also build a real world application with full C.R.U.D. operations (fig. 1.1.3).

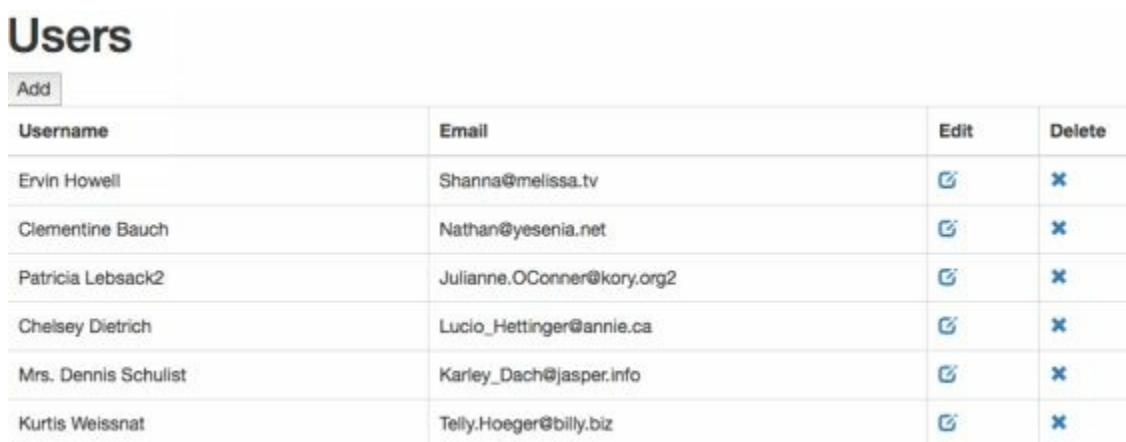


figure 1.1.3

These are the patterns you see on a lot of real world applications. In this book, you will learn how to implement these patterns with Angular 2.

1.2 Architecture of Angular 2 Apps

The four key players in an Angular 2 app are components, directives, routers and services.

Components

At the very core, we have components. A component encapsulates the template, data and behavior of a view. It is actually more accurate to call it a view component. For example, if we want to build an application like Amazon, we can divide it into three components. The search bar component, sidebar component and products component (fig. 1.1.2). A real world application would typically consists of tens or hundreds of components.

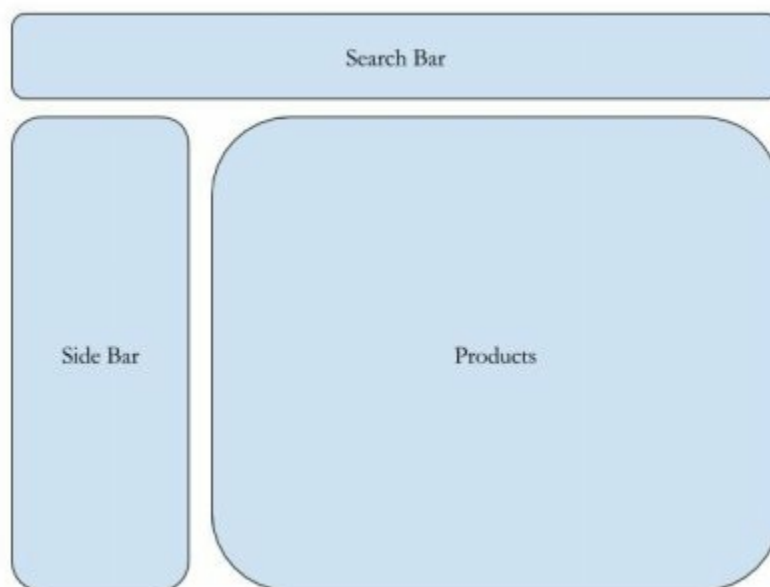


fig. 1.2.1

Each component will have its own html markup in its template as well as its own data and logic. Components can also contain other components. In products component where we display a list of products, we do so using multiple product components. Also, in each product component, we can have a rating component (fig. 1.2.2).

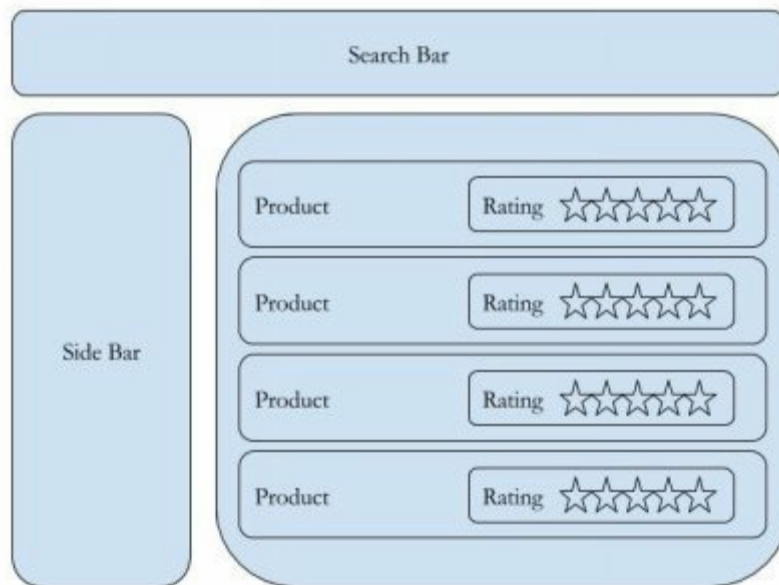


fig. 1.2.2

The benefit of such an architecture helps us to breakup a large application into smaller manageable components. Plus, we can reuse components within the application or even in a different application. For example, we can re-use the rating component in a different application.

What are components like in code? A component is nothing but a plain TypeScript class (see below code). Like any other class, it can have properties and methods. The properties hold the data for the view and the methods implement the behavior of a view, like what should happen if I click a button.

```
export class ProductComponent {  
  averageRating: number  
  setRating(value){  
    ...  
  }  
}
```

One thing that might be new for you if you have not worked with Angular 1 before is that these components are decoupled from the Document Object Model or DOM. In applications written with plain Javascript or JQuery, we get a reference to a DOM element in order to modify or handle its events. In Angular, we don't do that. Instead we use binding. In the view, we bind to the properties and methods of our components. We will cover binding in detail later in the book.

Services

Sometimes, our components need to talk to backend servers (e.g. Node, ASP.NET, Ruby on Rails) to get or save data. To have good separation of concerns in our application,

we delegate any logic that is not related to the user interface, to a ‘service’. A service is just a plain class that encapsulates any non user interface logic like making http calls, logging, business rules etc.

Router

The router is responsible for navigation. As we navigate from one page to another, it will figure out which component to present to the user based on changes in router name.

Directives

Similar to components, we use directives to work with the DOM. We use directives to add behavior to existing DOM elements. For example, we use the `autoGrow` directive to make the textbox automatically grow when it receives focus.

```
<input type="text" autoGrow />
```

Angular has a bunch of directives for common task like adding or removing DOM elements, adding classes or styles to them, repeating them. We can also create our own custom directives.

This is the big picture for components, services, router and directives. As you progress through this book, you will see each of these building blocks in action.

1.3 Getting the Tools

Installing Node

First, we need to install NodeJS. NodeJS is a server side language and we don’t need it because we are not writing any server side code. We mostly need it because of its npm or Node Package Manager. npm is very popular for managing dependencies of your applications. We will use npm to install other later tools that we need including Angular CLI.

Get the latest version of NodeJS from nodejs.org and install it on your machine. At this time of writing, we require at least NodeJS 4.x.x and npm 3.x.x. Installing NodeJS should be pretty easy and straightforward.

To check if Node has been properly installed, type the below on your command line (Command Prompt on Windows or Terminal on Mac):

```
node -v
```

and you should see the node version displayed.

To see if npm is installed, type the below on your command line:

```
npm -v
```

and you should see the npm version displayed.

Installing TypeScript

As explained, we will be using TypeScript for our Angular 2 development. To install TypeScript, type the following command:

```
npm install -g typescript
```

or

```
sudo npm install -g typescript
```

if you are a Mac user.

With -g specified, we install TypeScript globally on our machine so that we can use it no matter which folder we navigate too. The same applies for our other installations.

Installing Typings

Once TypeScript is installed, we install Typings. Typings is a module that allows us to bring in Javascript libraries into TypeScript. We will learn more about it later. So in Command Prompt type

```
npm install -g typings or sudo npm install -g typings for Mac users.
```

Installing Angular CLI

Angular CLI (Command Line Interface) is an official tool supported by the Angular 2 team which makes creating and managing Angular 2 projects simple. Setting up an Angular 2 project can be difficult on your own, but with the Angular CLI, it becomes much easier.

To install Angular CLI from the command line, type

```
npm install -g angular-cli
```

or for Mac,

```
sudo npm install -g angular-cli
```

TypeScript Editor

Next, we need a code editor that supports TypeScript. In this book, I will be using VScode (<https://code.visualstudio.com/>) which is a good, lightweight and cross platform editor from Microsoft. You can use Sublime, IntelliJ or any other editor that supports TypeScript.

Chrome Browser

Finally, I will be using Chrome as my browser. You can use other browsers but I highly recommend you use Chrome as we will be using Chrome developer tools in this book and I want to make sure you have the exact same experience as we go through the coding lessons.

1.4 Your First Angular 2 App

First, navigate to the folder where you want to create your Angular project. Next, simply create your Angular 2 project with the following command (ng refers to the Angular CLI tool),

```
ng new PROJECT_NAME
```

This will create your project folder and install everything you need to create your Angular 2 application. Note that this might take a couple of seconds.

When the folder is created, navigate to it by typing.

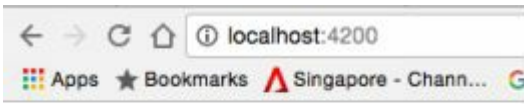
```
cd PROJECT_NAME
```

Next, type

```
ng serve
```

With the above command, the Angular 2 lite web server will start up on our machine so that we can run our application. It also compiles our TypeScript code back into Javascript since the browser is not able to run TypeScript.

Now, navigate to <http://localhost:4200/> and you should the message displayed as in fig.1.4.1.



app works!

fig. 1.4.1

Now let's look at our project files that have been created for us. When you open the project folder in VScode editor, you will find a couple of configuration files (fig. 1.4.2).



fig. 1.4.2

We will not go through all the files as our focus is to quickly get started with our first Angular 2 app, but we will briefly go through some of the more important files like `package.json`, `tsconfig.json`, `typings.d.ts`, `index.html` and the `app` folder which is the container for our application (the `app` folder is where we will work 99% of the time!). In the course of this book, you will come to appreciate the uses for the rest of the configuration files.

In the `src` folder, we have `main.ts` which is the starting module for our application. `ts` stands for TypeScript since this file is written in TypeScript. You will notice later that many of our other source files end with `.ts` since they are written in TypeScript

`tsconfig.json` is the configuration file for the TypeScript compiler. It determines how to transpile our TypeScript files into Javascript.

```
{
  "compilerOptions": {
    "declaration": false,
```

```

    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": ["es6", "dom"],
    "mapRoot": "./",
    "module": "es6",
    "moduleResolution": "node",
    "outDir": "../dist/out-tsc",
    "sourceMap": true,
    "target": "es5",
    "typeRoots": [
      "../node_modules/@types"
    ]
  }
}

```

For example in our tsconfig.json file above, we see in **bold** that the Javascript version the compiler transpiles to is 'es5' which is the current version of Javascript (at point of writing). To understand more information about the tsconfig file, it is best to look at tsconfig documentation on GitHub.

package.json is a standard node package configuration.

```

{
  "name": "angular2-firstapp",
  "version": "0.0.0",
  "license": "MIT",
  "angular-cli": {},
  "scripts": {
    "start": "ng serve",
    "lint": "tslint \"src/**/*.ts\"",
    "test": "ng test",
    "pree2e": "webdriver-manager update",
    "e2e": "protractor"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "2.0.0",
    "@angular/compiler": "2.0.0",
    "@angular/core": "2.0.0",
    "@angular/forms": "2.0.0",
    ...

```

You see in **bold** above the name of our application and its version.

In dependencies , we have the list of dependencies for our application.

```

"dependencies": {
  "@angular/common": "2.0.0",
  "@angular/compiler": "2.0.0",

```

```
"@angular/core": "2.0.0",
"@angular/forms": "2.0.0",
```

In the scripts section, we have a few custom node commands. If you run `npm start` it actually runs `ng serve`. `ng serve` runs the TypeScript compiler in ‘watch’ mode and starts our lite web server.

```
"scripts": {
  "start": "ng serve",
  "lint": "tslint \"src/**/*.ts\"",
  "test": "ng test",
  "pre2e": "webdriver-manager update",
  "e2e": "protractor"
},
```

With TypeScript compiler running in ‘watch’ mode, we can modify and save a TypeScript file, and the TypeScript compiler automatically watch for file changes and loads the changes in the browser. We will illustrate this later.

In the `app` folder, we find a couple of other TypeScript files:

app.module.ts

`app.module.ts` is the entry point to our application. An Angular application comprises of separate modules which are closely related blocks of functionality. Every Angular application has at least one module: the root module, named `AppModule` here. For many small applications, the root module `AppModule` alone is enough. For bigger modules, we can create multiple modules. We will illustrate this in **Chapter 11 - Structuring Large Apps With Modules**.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
```

```
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
```

```
bootstrap: [AppComponent]
}))
export class AppModule { }
```

A module is a plain TypeScript class with the `@NgModule` decorator. The decorator tells Angular that this class is going to be a module. This decorator adds metadata above this class. All modules and components in Angular are essentially decorated TypeScript classes. In the decorator are array attributes which Angular looks for in a module class:

declarations - to declare which components, directives or pipes are in this module. For now, it is just `AppComponent`. But we will soon start adding other components to this array.

imports - to specify what other modules do we use for this module. Angular 2 comes with other pre-defined modules like the `BrowserModule`, `FormsModule` and `HttpModule`. As a brief introduction, the `BrowserModule` contains browser related functionality. It also contains the common module which has `ngIf`, `ngFor` which we will introduce later. The `FormsModule` is needed when working with input fields and other forms related functionality. The `HttpModule` is needed when working with http access.

providers - to specify any application wide services we want to use

Since our application is a web application that runs in a browser, the root module needs to import the `BrowserModule` from `@angular/platform-browser` to the imports array. For now, our application doesn't do anything else, so you don't need any other modules. In a real application, you'd likely import `FormsModule` and `HttpModule` and that is why we keep it there.

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

Every Angular application has at least one component: the root component, named `AppComponent` in `app.component.ts`. Components are the basic building blocks of Angular applications. A component controls a portion of the screen, a view - through

its associated html template, app.component.html . app.component.css is the css file referenced from app.component.ts

For now, our root app component is nothing but a plain TypeScript class with a variable title ,

This class is decorated with the Component decorator @Component . Like the module decorator, the component decorator adds metadata above this class. Because decorators are functions, we need to use the prefix @ sign to call the @Component function with its brackets @Component(...) . All components in Angular are essentially decorated TypeScript classes.

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

1.5 TypeScript Compilation

Because our TypeScript compiler is running in ‘watch’ mode, our app will automatically reload if we change any of the source files. To illustrate this, change the value of ‘title’ variable to

```
export class AppComponent {  
  title = 'My Second Angular App!';  
}
```

and save the file.

Because TypeScript compiler is running in the ‘watch’ mode, it detects that there is a file change and re-compiles the file. If you switch back to your Chrome browser, the app gets refreshed automatically so you don’t have to refresh the page every time your code changes. On your browser, you should see something like below.

My Second Angular App

How does this happen? The html markup of our component is stored in app.component.html which has the following code

```
<h1>  
  {{title}}  
</h1>
```

With `{{title}}` we are using string interpolation which we will explore this in detail later. But basically, this allows you to output any property of your component dynamically into the html. So `{{title}}` here actually refers to the title property in our `app.component.ts` .

Summary

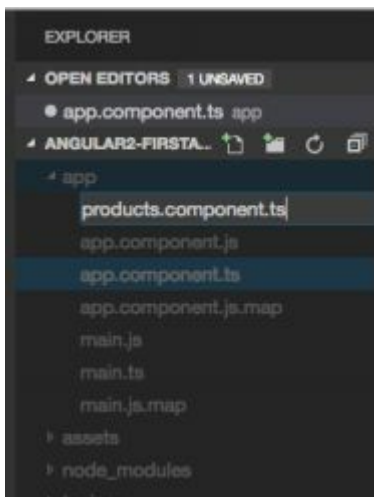
In this chapter, we have been introduced to the core building blocks of Angular 2 apps which are components, directives, services and routers. We have also been introduced to the Angular 2 development experience which is coding in TypeScript and having TypeScript compiler automatically generate our app for us that we can view on the browser. In the next chapter, we will begin implementing an Angular 2 app.

CHAPTER 2: ANGULAR 2 QUICKSTART

In the previous chapter, you learned about the core building blocks of Angular 2 apps, components, directives, services and routers. In this chapter, we will implement a component, directive and service from scratch to have an idea on what it is like to build an Angular 2 app.

2.1 Creating Components

In VSCode, open the project folder that you have created in chapter 1. We first add a new file in the app folder and call it `products.component.ts` .



Note the naming convention, we start with the name of the component products followed by `component.ts` . Remember that `ts` stands for TypeScript.

Type out the below code into `products.component.ts`:

```
import { Component } from '@angular/core'
```

```
@Component({  
  selector: 'products',  
  template: '<h2>Products</h2>'  
})  
export class ProductsComponent {  
  
}
```

Code Explanation

import { Component } from '@angular/core' imports the component decorator from the core Angular module.

The component decorator @Component tells Angular that this class is going to be a component.

The @Component function takes in an object {} with 2 attributes, selector and template both of type string as shown below:

```
@Component({
  selector: 'products',
  template: '<h2>Products</h2>'
})
```

selector specifies a css selector for a host html element. When Angular sees an element that matches this css selector, it will create an instance of our component in the host element. Here, our host element is an element with tag, products .

template specifies the html that will be inserted into the DOM when the component's view is rendered. We can either write the html here inline, or reference to it in a separate html file with the attribute templateUrl which we see in app.component.ts . Our currenthtml markup is <h2>Products</h2>.

Lastly, the export keyword makes this class available for other files in our application to import this class.

With these simple lines of code, we have just built our first component!

2.2 Using Components

Now, go back to app.component.ts . In the VScode editor, you can use the 'Ctrl-P' or 'Command-P' shortcut on Mac to quickly navigate to files by typing in the filename. Else, you can just navigate to your file using the navigation directory on the left.

You should by now notice that the contents of app.component.ts is very similar to products.component.ts .

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
title = 'My Second Angular App';  
}
```

To re-iterate, we import the component decorator, import { Component } from '@angular/core';

We then call it using @Component() and give it an object with fields, selector, templateUrl and styleUrls. With the field templateUrl, we refer to our html markup in a separate file app.component.html. When the html markup is big, it is better to put it in a separate file using templateUrl. However, if the html markup is not too big, it is easier to have it in the component.ts file as we can view it all at a single glance. Let's do this now by changing templateUrl to template and inserting our html markup in app.component.ts. Change your code to the below:

```
@Component({  
  selector: 'app-root',  
  template: '<h1>{{title}}</h1>',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  title = 'My Second Angular App';  
}
```

The same applies for the styleUrls attribute, which refers to app.component.css as the css file. Because app.component.css is currently empty, and we have no use of it yet, we can just remove that attribute to make the code cleaner as in below (remember to remove the comma at the end of the template line):

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-root',  
  template: '<h1>{{title}}</h1>'  
})  
export class AppComponent {  
  title = 'My Second Angular App';  
}
```

At this point, we can actually delete app.component.html and app.component.css file.

Lastly, we export this component in export class AppComponent. Remember that App component is the root of our application. It is the view component that controls our entire app or page.

Now, add <products></products> to the template as shown below.

```
import { Component } from '@angular/core';
```

```
@Component({  
  selector: 'app-root',  
  template:  
    `<h1>{{title}}</h1>  
    <products></products>`  
})  
export class AppComponent {  
  title = 'My Second Angular App';  
}
```

```
export class AppComponent { }
```

*Do note that the slashes used in the template attribute is the **backtick** character ```. The backtick ``` is located in the top left of the keyboard just before the number 1. The backtick allows us to put html markup into multiple lines.

app.module.ts

Next, edit the file `app.module.ts` to import your new `ProductsComponent` and add it in the declarations array in the `NgModule` decorator (see below code in **bold**). Because `ProductsComponent` is added to the declarations array of `AppModule`, `AppComponent` does not have to import `ProductsComponent` again since we have specified that `AppComponent` and `ProductsComponent` belong to the same module and therefore have access to one another.

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/http';
```

```
import { AppComponent } from './app.component';
```

```
import { ProductsComponent } from './products.component';
```

```
@NgModule({  
  declarations: [  
    AppComponent, ProductsComponent  
  ],  
  imports: [  
    BrowserModule,  
    FormsModule,  
    HttpClientModule,  
  ],  
  providers: [],  
  bootstrap: [AppComponent]
```

```
  })  
  export class AppModule { }
```

Save the file and go to your browser. You should see the Products component markup displayed as in fig. 2.2.1.

My Second Angular App

Products

fig. 2.2.1

Code Explanation

We first import our ProductsComponent using

```
import { ProductsComponent } from './products.component';
```

For custom components that we have defined, we need to specify their path in the file system. Since App component and ProductsComponent are in the same folder app , we use './' which means start searching from the current folder followed by the name of the component, products.component (without .ts extension).

When we add ProductsComponent to the declarations array, we are saying that it is part of this module. So when Angular sees the <products> tag, Angular will know that ProductsComponent is responsible for that.

```
template: `  
  <h1>{{title}}</h1>  
  <products></products>  
,`
```

<products></products> here acts as a custom directive. Remember that a directive is a class that allows us to extend or control our Document Object Model. In this way, we can design custom elements that are not part of standard html. In our case, we use the ProductsComponent to define a new element. Every component is technically a directive. The difference is that a component has a template and a directive doesn't.

If we inspect the html element (fig. 2.2.2), we see that the root element is <app-root> having two child elements <h1>My Second Angular App</h1> and <products> . <products> has in turn <h2>Products</h2> .

```

<html>
  <head>...</head>
  <body>
    <app-root>
      <h1>My Second Angular App</h1>
      <products>
        <h2>Products</h2>
      </products>
    </app-root>
    <script type="text/javascript" src=
    <script type="text/javascript" src=
    <script type="text/javascript" src=
  </body>
</html>

```

fig. 2.2.2

This is because our root component's template is defined as

```

@Component({
  selector: 'app-root',
  template:
    `<h1>{{title}}</h1>
    <products></products>`
})

```

And why is `<app-root>` rendered? If we look at `index.html` below, we see the element `<app-root>` referenced in `<body>`. Angular saw this and rendered the App component.

```

<html>
<head>
  <meta charset="utf-8">
  <title>Angular2Firstapp</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>

```

2.3 Templates

Because a component encapsulates the data and the logic behind the view, we can define properties and display them in the template. For example, App component has a property `title` which holds the value 'My Second Angular App'.

```

export class AppComponent {
  title = 'My Second Angular App';
}

```

Note that Angular can automatically infer the type of the variable by the value assigned

to it. But we can explicitly set the type of the variable (if we want to) as shown below.

```
title: string = 'My Second Angular App';
```

Interpolation

The value of title is rendered on to our template with doubly curly braces `{{title}}`.

```
template:
`<h1>{{title}}</h1>
<products></products>`
```

This is called interpolation. If the value of this property in the component changes, the view will be automatically refreshed. This is called one way binding (where the value displayed in the html markup is binded to the component's property). We also have two way binding which is used in forms. For example, when we type something into an input field that is bound to a property, as you modify the value of the input field, the component's property will be updated automatically. You will see that later in the Forms chapter.

Displaying a List

We will illustrate using properties further by displaying a list of products in ProductsComponent. First, we declare an array products in ProductsComponent which contains the names of the products that we are listing.

```
export class ProductsComponent{
  products = ["Learning Angular 2", "Pro TypeScript", "ASP.NET"];
}
```

Next in the template, we use `` and `` to render the list of products.

```
@Component({
  selector: 'products',
  template: `
    <h2>Products</h2>
    <ul>
      <li *ngFor="let product of products">
        {{product}}
      </li>
    </ul>
  `,
})
```

Navigate to your browser and you should see the result in fig. 2.3.1

My Second Angular App

Products

- Learning Angular 2
- Pro TypeScript
- ASP.NET

fig. 2.3.1

Code Explanation

`<li *ngFor="let product of products">` allows us to repeat `` for each product. The `*ngFor` keyword acts like a for-loop, where we let product be the temporary local variable representing each element in the array products .

For each product, we use interpolation `{{product}}` to display the product.

Templates in Angular looks similar to html most of the time. Sometimes, we use special attributes like `*ngFor` which is an example of a directive provided by Angular. This directive extends the html and adds extra behavior, in this case, repeating the `` element based on the expression assigned to it.

Services and Dependency Injection

Currently, our component is rendering a hardcoded list of products. In a real world application however, we would get the data from a server. Components should only contain logic related to the view. Logic to get data from a server should not be in a component but instead be encapsulated in a separate class called a **service**. We will now create a service that will get data from a server.

In the `app` folder, create a new file called `product.service.ts` . Note the naming convention, we start with the name of the service and then `' .service.ts '` . Type in the below code into `' product.service.ts '` .

```
export class ProductService{
  getProducts() : string[] {
    return ["Learning Angular 2","Pro TypeScript","ASP.NET"];
  }
}
```

Code Explanation

If you notice, a service (like a component) is just a plain class. It contains a

method `getProducts()` of return type string array as represented by `getProducts():string[]` . For now, we illustrate by returning the same hard-coded array as before so as not to get distracted by how to call a RESTful api in our service (we will do that later in chapter 8 and 9).

Dependency Injection

Back in `products.component.ts` , change the code to below.

```
export class ProductsComponent{
  products;

  constructor(productService: ProductService){
    this.products = productService.getProducts();
  }
}
```

Code Explanation

Our string array `products` is populated by the constructor. The constructor takes in a `ProductService` object. We call the `getProducts()` method of the `ProductService` object and assign the results to our `products` array.

You might ask, how do we create the `Product` service and pass it into the constructor? That's when Dependency Injection which is built into Angular comes into the picture. Dependency injection injects dependencies of your classes when creating them. So when creating a `ProductsComponent` , it looks at the constructor and see that we need a `ProductService` , it will create an instance of `ProductService` and then inject it into the constructor of the `ProductsComponent` class.

Finishing Up

Lastly, in `app.component.ts` , add the lines in **bold**.

```
import { Component } from '@angular/core';

import { ProductService } from './product.service';

@Component({
  selector: 'app-root',
  template:
    `

# {{title}}

</h1>
    <products></products>`,
  providers: [ProductService]
})
```

```
export class AppComponent {  
  title: string = 'My Second Angular App';  
}
```

The providers array contain the dependencies of the ProductsComponent . So here, we say that ProductService is a dependency of ProductsComponent .

If you run the application now, you should see the same list of products as before. Only now that we have encapsulated the data retrieval logic to ProductService and ensure that ProductsComponent contains only user-interface related logic.

Try This

You have learnt a lot in this chapter. Now to crystalize all that you have learnt, try to do the following task. Extend this application to have a AdvertisementsComponent which displays a list of advertisements below the products. You can do this by going through the same steps as we did. First, create AdvertisementsComponent and AdvertisementService . Then, use the *ngFor and interpolation {{ }} to render the list of advertisements.

It would be beneficial to do this exercise as we are going on to more complex topics in the following chapter and I want to make sure that you master the fundamental topics before we get there. If you get stuck or if you would like to get the sample code we have used in this chapter, contact me at support@i-ducate.com.

Summary

In this chapter, we briefly looked at Components, Directives, Services and Dependency Injection. We have created a ProductsComponent that retrieves product data through a Service, and later displays that data on the page.

CHAPTER 3: RENDERING DATA AND HANDLING EVENTS

In this chapter, we will explore different kinds of bindings in Angular 2 apps like binding various properties of DOM elements to component properties, how to apply css classes on styles dynamically and how to handle events raised from DOM elements.

3.1 Property Binding

We have learnt about interpolation where we display properties of a component in the view for example, `<h1>{{title}}</h1>`. Interpolation can also be applied to other elements like `` to display an image in the view as shown below.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    
  `
})
export class AppComponent {
  title = 'My Second Angular App';
  imageUrl = "http://lorempixel.com/400/200/";
}
```

In the code above, using interpolation, we bind the `src` property of our DOM element to the `imageUrl` property in component. When there are changes in the `imageUrl` property of the component, it will be reflected in the DOM. Any changes in the DOM however are not reflected in the component. This is one way binding.

There are also two alternative syntaxes to bind properties.

```
<img [src]="imageUrl" />

```

For the first alternative, we put the DOM property in square brackets `[]`. For the second alternative, we prefix it with `bind-`. Although all three syntaxes are identical, it is preferable to use

`` as it is cleaner.

3.2 CSS Class Binding

In the below code, we show a button in our view using two bootstrap css classes `btn` and `btn-primary` to make our button look more professional. If you are not familiar with bootstrap, it is an html, css, javascript framework to help build user interface components for websites or web applications. It contains html and css-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. For now, we are interested in its templates for buttons to make them look better.

```
@Component({
  selector: 'app-root',
  template: `
    <button class="btn btn-primary">Submit</button>
  `
})
```

We first however need to reference `bootstrap.css` in our `index.html`. Go to getbootstrap.com and under 'Getting Started', copy the stylesheet link (3.2.1),



fig. 3.2.1

and paste the link into `index.html` as shown below in **bold**.

```
<html>
<head>
  <meta charset="utf-8">
  <title>Angular2Firstapp</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/.../bootstrap.min.css" ...>
</head>
<body>
  <app-root>Loading...</app-root>
```

```
</body>
</html>
```

If you have successfully linked your bootstrap class, you should get your button displayed like in fig. 3.2.2.

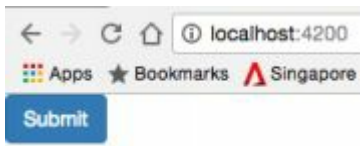


fig. 3.2.2

There are times when we want to use different css classes on an element based on different conditions. For example, if I want to add the active class to the button based on varying conditions, I can do the following

```
@Component({
  selector: 'app-root',
  template: `
    <button
      class="btn btn-primary"
      [class.active]="isActive">Submit</button>
  `,
})
export class AppComponent {
  isActive = true;
}
```

That is, when `isActive = true` the active css class will be applied to the button making it slightly darker.



If `isActive = false` the active css class will **not** be applied to the button making it lighter.



3.3 Style Binding

Style binding is a variation of property binding, similar to css class binding. Style binding applies inline styles to our button for example:

```
@Component({
  selector: 'app-root',
  template: `
    <button
```

```

class="btn btn-primary"
[style.backgroundColor]="isActive ? 'blue':'gray'">Submit</button>
,
}))
export class AppComponent {
  isActive = true;
}

```

In the above code, we bind style property backgroundColor to blue if isActive is true and gray if isActive is false.

 if isActive is true

 if isActive is false

3.4 Event Binding

We use event binding to handle events raised by the DOM like clicks, mouse movements, key strokes and so on. Similar to property binding, we have two syntaxes for event binding. parenthesis () or prefix.

```

@Component({
  selector: 'app-root',
  template: `
    <button (click)="onClick($event)">Submit</button>
  `,
})
export class AppComponent {
  onClick($event){
    console.log("Clicked",$event)
  }
}

```

In the above code, we illustrate event binding using **parenthesis**. We put the target event name (click) name in the parenthesis and then assign it to the onClick(\$event) method in our component.

Alternatively, we can also have,

<button on-click="onClick(\$event)">Submit</button> where we add the prefix on- before the target event.

When the button is clicked, the onClick method is called. The \$event argument in onClick(\$event) allows us to get access to the event raised. For example, in mouse movements, the event object will tell us the x and y position. We can also use the event object to get access to the DOM element that raised the event. Note that there is

a \$ prefix in \$event because this is built into Angular. The event object is a standard DOM event object and has got nothing to do with Angular.

3.5 Two-way Binding

In Angular, there is a directive called `ngModel` to create two way binding between a component property and a DOM property in the view. Remember that a directive is a class that allows us to control or extend the behavior of a DOM. We illustrate two way binding using `ngModel` in the below code.

```
@Component({
  selector: 'app-root',
  template: `
    <input type="text" [(ngModel)]="title" />
    You have typed: {{title}}
  `,
})
export class AppComponent {
  title = "hello";
}
```

When you run the app, whatever you type in the input field is displayed in the DOM as well (fig. 3.5.1).



fig. 3.5.1

Code Explanation

In the input field, we add `[(ngModel)]="title"` to bind our DOM input property to the component's title property. But realize that the component property is also binded to the DOM in the code `You have typed: {{title}}`

3.6 Example Application

We will now put into practise what we have learnt about data binding. We will build a rating component like in figure 3.6.1.



fig. 3.6.1

You have seen such a rating component in many places e.g. Amazon. So we can implement this as a component and reuse it in many places. A user can click select from a rating of one star to five stars. For now, don't worry about calling a server or any other logic. We just want to implement the UI.

Open the existing project folder from chapter two, create a new component class `rating.component.ts` and fill it with the below code.

```
import { Component } from '@angular/core'

@Component({
  selector: 'rating',
  template: `
    <i
      class="glyphicon"
      [class.glyphicon-star-empty]="rating < 1"
      [class.glyphicon-star]="rating >= 1"
      (click)="onClick(1)"
    >
    </i>
  `,
})
export class RatingComponent {
  rating = 0;

  onClick(ratingValue) {
    this.rating = ratingValue;
  }
}
```

Next in `app.module.ts`, import the rating component and add it to the `declarations` array by adding the below lines in **bold**. By adding `RatingComponent` to `declarations`, we specify that it is part of `AppModule` and can therefore be used in `AppComponent`.

```
import { ProductsComponent } from './products.component';
import { RatingComponent } from './rating.component';

import { ProductService } from './product.service';

@NgModule({
  declarations: [
    AppComponent, ProductsComponent, RatingComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
  ],
  //providers: [ProductService],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Lastly, in app.component.ts , add the <rating> element in the template as shown below:

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <rating></rating>
  `
})
export class AppComponent {
  title = "hello";
}
```

Code Explanation

```
@Component({
  selector: 'rating',
  template: `
    <i
      class="glyphicon"
      [class.glyphicon-star-empty]="rating < 1"
      [class.glyphicon-star]="rating >= 1"
      (click)="onClick(1)"
    >
    </i>
  `
})
```

We define a component called Rating Component with selector as 'rating' so in app.component.ts , we create an instance of it with <rating></rating>.

In the template, we render bootstrap glyphs using class="glyphicon" and use class binding to conditionally render a secondary class with

```
[class.glyphicon-star-empty]="rating < 1"
[class.glyphicon-star]="rating >= 1"
```

The condition is based on the property rating defined in the RatingComponent class below.

```
export class RatingComponent{
  rating = 0;

  onClick(ratingValue){
```

```

        this.rating = ratingValue;
    }
}

```

The condition works like the following, if the rating is less than one, we render the empty star icon.

```
[class.glyphicon-star-empty]="rating < 1"
```

If the rating is more than or equal to one, we render the normal star icon.

```
[class.glyphicon-star]="rating >= 1"
```

We add a click handler (click)="onClick(1)" to assign a rating of one if a user clicks on this star.

You would probably notice that we have only one star at this point of time, when ratings usually have five stars. This is because I wanted you to be familiar with the logic of one star. To then extend it to five stars is easy. Simply copy and paste the glyphicon code in the template for additional stars as shown below.

```

@Component({
  selector: 'rating',
  template: `
    <i
      class="glyphicon"
      [class.glyphicon-star-empty]="rating < 1"
      [class.glyphicon-star]="rating >= 1"
      (click)="onClick(1)"
    >
  </i>
  <i
    class="glyphicon"
    [class.glyphicon-star-empty]="rating < 2"
    [class.glyphicon-star]="rating >= 2"
    (click)="onClick(2)"
  >
  </i>
  <i
    class="glyphicon"
    [class.glyphicon-star-empty]="rating < 3"
    [class.glyphicon-star]="rating >= 3"
    (click)="onClick(3)"
  >
  </i>
  <i
    class="glyphicon"
    [class.glyphicon-star-empty]="rating < 4"
    [class.glyphicon-star]="rating >= 4"

```

```

        (click)="onClick(4)"
    >
</i>
<i
    class="glyphicon"
    [class.glyphicon-star-empty]="rating < 5"
    [class.glyphicon-star]="rating >= 5"
    (click)="onClick(5)"
>
</i>
,
}))

```

Note that you need to do two things. Firstly, change the value of each condition depending on which star it is. The second star's condition should be

```

[class.glyphicon-star-empty]="rating < 2"
[class.glyphicon-star]="rating >= 2"

```

The second star should be empty if the rating is less than two. It should be filled if the rating is more than or equal to two. The same goes for the third, fourth and fifth star.

Secondly, change the value of the argument when you called the `onClick` method depending on which star it is. The second star's `onClick` should be `(click)="onClick(2)"`. So when a user clicks on the second star, the `onClick` method is called with property rating of value two. When a user clicks on the third star, the `onClick` method is called with property rating of value three and so on.

Summary

In this chapter, you learnt about property binding, class binding, style binding, event binding, and two way binding. In the next chapter, we will take a closer look at Angular components.

Contact me at support@i-ducate.com if you have not already to have the full source code for this chapter or if you encounter any errors with your code.

CHAPTER 4: BUILDING RE-USABLE COMPONENTS

In this chapter, we will learn more about components and how to reuse them in an application.

4.1 Component Input Properties

We can mark properties in our component as input or output properties. In doing so, we are defining a public API for our component. Properties marked as input or output will be visible from the outside and available for property or event binding. For example, in our button element, we can bind its value property to a property in our component.

```
<button [value] = "title" (click)="onClick($event)">Submit</button>
```

Value here is an example of an input property. We can use it to pass data to our button. Buttons also have events like (click) that we can bind to methods in our components. (click) is an example of an output property. The input and output properties form the API of a button.

We can also define the public api for a custom component. Why would we want to do that? Suppose we want to display a list of products with its rating. We will need to assign the rating value to our rating component beforehand. However, we can only use our rating component like this `<rating></rating>` now.

If we want to do something like, `<rating [rating]="4"></rating>` to display a rating of 4 stars, we have to declare our component property rating with the `@Input` decorator. And that is the purpose of public APIs.

To declare an input property, first we need to import `Input` (see code in **bold** below).

```
import { Component, Input } from '@angular/core'
```

Next, add the `@Input()` decorator before the property as shown below.

```
@Component({  
  selector: 'rating',  
  template: `  
    ...  
  `,  
})  
export class RatingComponent{
```

```

@Input() rating = 0;

onClick(ratingValue){
  this.rating = ratingValue;
}
}

```

Aliasing

If you want to expose the property using a different name, for example `<rating [rating-value]="4"></rating>`, you can do this by supplying the new name to the `@Input` function like `@Input('rating-value') rating = 0;`

4.2 Templates

Till now, we have been writing our template inline with our component.

```

@Component({
  selector: 'rating',
  template: `
    <i
      class="glyphicon"
      [class.glyphicon-star-empty]="rating < 1"
      [class.glyphicon-star]="rating >= 1"
      (click)="onClick(1)"
    >

```

If a template code is small enough, it is better to define it in our component rather than in a separate file. Why? If we want to reuse this component, we simply take the file and we are done. The component is self contained. But if the template is in a separate file, we have to remember to copy both files.

If a template gets large however, we shouldn't write our templates inline. We should move it to a separate file for better separation of concerns. In our rating component, notice that the template can get quite big.

To put the template in a separate file, we create a new file, `rating.component.html` in the app folder. Cut and paste the template html from `rating.component.ts` into `rating.component.html`.

In rating.component.ts , make the changes as highlighted in **bold** below.

```
import { Component, Input } from '@angular/core'
```

```
@Component({  
  selector: 'rating',  
  templateUrl: 'rating.component.html'  
})  
export class RatingComponent {  
  @Input('rating-value') rating = 0;  
  
  onClick(ratingValue) {  
    this.rating = ratingValue;  
  }  
}
```

Instead of having our html in template , we specify the url to our template file in templateUrl . At the risk of sounding obvious, do note that you cannot use both template and templateUrl at the same time.

4.3 Styles

Another useful field in the component metadata is styles or styleUrls . Similar to template , you can define css styles required by your component in the styles array or in a separate file(s) with styleUrls . A notable feature is that these styles are scoped to your component. They won't effect to the outer html or other components.

To illustrate, suppose we want our filled glyphicon stars to be orange, we add the following in **bold**.

```
@Component({  
  selector: 'rating',  
  templateUrl: 'rating.component.html',  
  styles: [  
    .glyphicon-star {  
      color: orange;  
    }  
  ]  
})
```

When we run our application, we will see our filled stars with the orange css applied to

4.4 Example Application

We will reuse the rating component that we have made and implement a product listing like in figure 4.4.1.

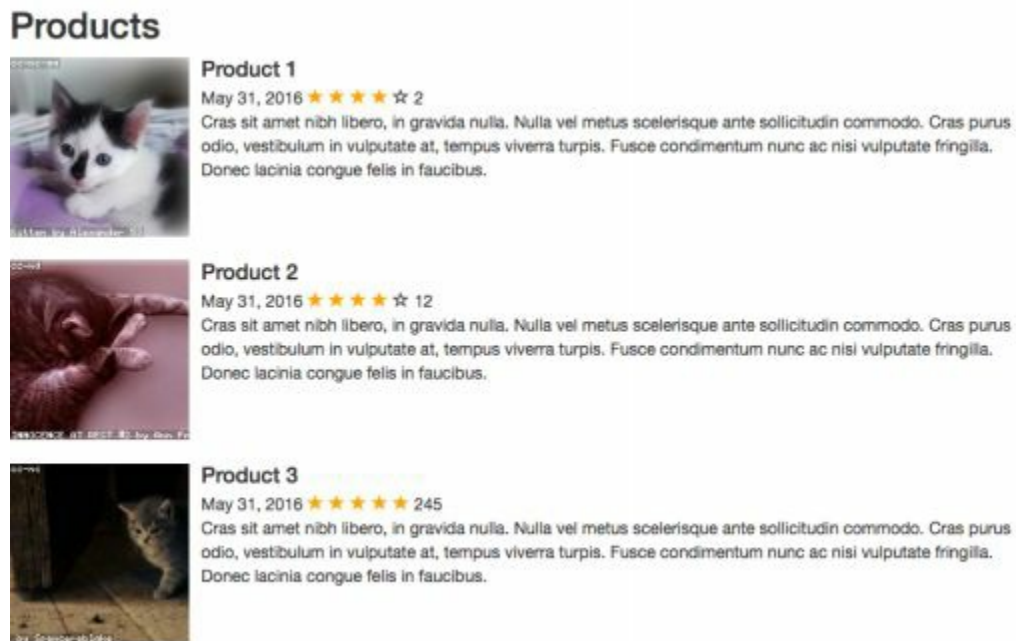


fig. 4.4.1

This is like the list of products on Amazon. For each product, we have an image, the product name, the product release date, the rating component and the number of ratings it has.

In the same project folder you have used to create the rating component, create a new component file `product.component.ts` that contains `ProductComponent`. This component will be used to render one product. Fill in the file with the below code.

```
import {Component, Input} from '@angular/core';
```

```
@Component({
  selector: 'product',
  template: `
    `,
  styles: [
    `.media {
      margin-bottom: 20px;
    }
  `
])
export class ProductComponent {
```

```

    @Input() data;
}

```

Now, how do we get our template to render each product listing like in figure 4.4.1? We use the media object in bootstrap. Go to getbootstrap.com, in the **components** page, click on media object and copy the markup there into the template field of ProductComponent.



fig. 4.4.2

Next in the template, we use interpolation to assign values of our product into the DOM. Type in the below codes in bold into the template.

```

@Component({
  selector: 'product',
  template: `
    <div class="media">
      <div class="media-left">
        <a href="#">
          
        </a>
      </div>
      <div class="media-body">
        <div class="media-body">
          <h4 class="media-heading">
            {{ data.productName }}
          </h4>
          {{ data.releasedDate }}
          <rating
            [rating-value]="data.rating"
            [numOfReviews]="data.numOfReviews">
          </rating>
          <br>
          {{ data.description }}
        </div>
      </div>
    </div>
  `
})

```

```

    </div>
  ,
  styles: [
    .media {
      margin-bottom: 20px;
    }
  ]
})

```

With the above code, our product component is expecting a data object with the fields: imageUrl, productName, releasedData and description .

We have also added our rating component that expects input rating and number of reviews.

```

    <rating
      [rating-value]="data.rating"
      [numOfReviews]="data.numOfReviews">
    </rating>

```

Our rating component currently only has rating-value as input. Add the below code in **bold** into rating.component.ts to add numOfReviews as input.

```
import { Component, Input } from '@angular/core'
```

```

...
export class RatingComponent{
  @Input('rating-value') rating = 0;
  @Input() numOfReviews = 0;

  onClick(ratingValue){
    this.rating = ratingValue;
  }
}

```

Next, we create a new file product.service.ts that contains a service class ProductService that is responsible for returning a list of products. Type in the below code into ProductService class.

```

export class ProductService{
  getProducts() {
    return [
      {
        imageUrl: "http://loremflickr.com/150/150?random=1",
        productName: "Product 1",
        releasedDate: "May 31, 2016",
        description: "Cras sit amet nibh libero, in gravida... ",
        rating: 4,

```

```

        numOfReviews: 2
      },
      {
        imageUrl: "http://loremflickr.com/150/150?random=2",
        productName: "Product 2",
        releasedDate: "October 31, 2016",
        description: "Cras sit amet nibh libero, in gravida... ",
        rating: 2,
        numOfReviews: 12
      },
      {
        imageUrl: "http://loremflickr.com/150/150?random=3",
        productName: "Product 3",
        releasedDate: "July 30, 2016",
        description: "Cras sit amet nibh libero, in gravida... ",
        rating: 5,
        numOfReviews: 2
      }
    ];
  }
}

```

Notice that in our class, we currently hardcode an array of product objects. Later on, we will explore how to receive data from a server.

For imageUrl , we use <http://loremflickr.com/150/150?random=1> to render a random image 150 pixels by 150 pixels. For multiple product images, we change the query string parameter random=2, 3,4 and so on to get a different random image.

Just like before, the getProduct method in ProductService will be called by ProductsComponent . The code remains largely the same (see below) and that is the benefit for having separation of concerns to have data retrieving functionality (non UI related) in a separate service class rather than in a component.

```

import { Component } from '@angular/core'
import { ProductService } from './product.service'

@Component({
  selector: 'products',
  template: `
    <h2>Products</h2>
    <div *ngFor="let product of products">
      <product [data]="product"></product>
    </div>
  `,
  providers: [ProductService]
})
export class ProductsComponent {

```

```

products;

constructor(productService: ProductService){
    this.products = productService.getProducts();
}
}

```

The ngFor loops through products array as retrieved from ProductService and inputs each data element in the products array for each product component. Each data element provides Product component with values from properties imageUrl, productName, releasedData and description .

We also import ProductService and declare it in the providers array to specify that we depend on ProductService as a service provider.

Lastly in app.module.ts in the lines in **bold** below, we import ProductComponent and add it to the declarations array to declare ProductComponent as part of AppModule .

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';

import { ProductsComponent } from './products.component';
import { ProductComponent } from './product.component';
import { RatingComponent } from './rating.component';

import { ProductService } from './product.service';

@NgModule({
  declarations: [
    AppComponent, ProductsComponent, RatingComponent, ProductComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Save all your files and you should have your application running fine like in figure 4.4.3.

Products



Product 1

May 31, 2016 ★★★★★ ☆ 2

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo. Cras purus odio, vestibulum in vulputate at, tempus viverra turpis. Fusce condimentum nunc ac nisi vulputate fringilla. Donec lacinia congue felis in faucibus.



Product 2

May 31, 2016 ★★★★★ ☆ 12

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo. Cras purus odio, vestibulum in vulputate at, tempus viverra turpis. Fusce condimentum nunc ac nisi vulputate fringilla. Donec lacinia congue felis in faucibus.



Product 3

May 31, 2016 ★★★★★ ☆ 245

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo. Cras purus odio, vestibulum in vulputate at, tempus viverra turpis. Fusce condimentum nunc ac nisi vulputate fringilla. Donec lacinia congue felis in faucibus.

figure 4.4.3

Summary

In this chapter, we illustrate how to define input properties in a component, use both inline templates and templates defined in a separate file with `templateUrl`, use styles to define css styles used in a component and borrow markup from *bootstrap* to put all these together in our example Product Listing application.

Contact me at support@i-ducate.com if you encounter any issues or if you had not requested the full source code for this chapter.

CHAPTER 5: CONTROLLING RENDERING OF HTML

We have worked with `ngFor` which is one of the built in directives in Angular 2. In this chapter, we will explore more built in directives which will give us more control in rendering html.

5.1 ngIf

Suppose you want to show or hide part of a view depending on some condition. For example, we have earlier displayed our list of products. But if there are no products to display, we want to display a message like “No products to display” on the page.

In `products.component.ts` , we add the codes in bold

```
import { Component } from '@angular/core'
import { ProductService } from '../product.service'

@Component({
  selector: 'products',
  template: `
    <h2>Products</h2>
    <div *ngIf="products.length > 0">
      <div *ngFor="let product of products">
        <product [data]="product"></product>
      </div>
    </div>
    <div *ngIf="products.length == 0">
      No products to display
    </div>
  `,
  providers: [ProductService]
})
export class ProductsComponent {
  products;

  constructor(productService: ProductService) {
    this.products = productService.getProducts();
  }
}
```

Now when we run our app again, we should see the products displayed as same as before. But if we comment out our hard-coded data in `ProductService` and return an

empty array instead, we should get the following message.

Products

No products to display

Code Explanation

```
<div *ngIf="products.length > 0">
  <div *ngFor="let product of products">
    <product [data]="product"></product>
  </div>
</div>
```

We used the `*ngIf` directive to add a `if`-condition in our DOM and assign an expression “`products.length > 0`” to it. If the expression evaluates to true, the `div` element and its children will be inserted into the DOM. If it evaluates to false, it will be removed from the DOM. When there are products returned from `ProductService`, the expression evaluates to true and renders the products in the `ngFor` loop.

The following expression however evaluates to false and therefore, we don’t display the message.

```
<div *ngIf="products.length == 0">
  No products to display
</div>
```

When we return an empty array however, “`products.length > 0`” evaluates to false and we do not render the list of products. Instead we display the “No products to display message”.

5.2 ngSwitch

In a similar fashion as `ngIf`, the `ngSwitch` statement allows us to render elements and its children based on a condition against a list of values.

Suppose we want to add a comment based on the product rating as in figure 5.2.1,

Products



Product 1

May 31, 2016 ★ ★ ★ ★ ☆ 2
Very Good

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus sceler
odio, vestibulum in vulputate at, tempus viverra turpis. Fusce cc
Donec lacinia congue felis in faucibus.



Product 2

October 31, 2016 ★ ★ ☆ ☆ ☆ 12
Fair

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus sceler
odio, vestibulum in vulputate at, tempus viverra turpis. Fusce cc
Donec lacinia congue felis in faucibus.



Product 3

July 30, 2016 ★ ★ ★ ★ ★ 2
Excellent

Cras sit amet nibh libero, in gravida nulla. Nulla vel metus sceler
odio, vestibulum in vulputate at, tempus viverra turpis. Fusce cc
Donec lacinia congue felis in faucibus.

fig. 5.2.1

where one star means ‘Poor’, two star - ‘Fair’, three star - Good, four star - ‘Very Good’, five star - ‘Excellent’. We can achieve this by adding the below code in **bold** into the template field of product.component.ts .

template: `

```
<div class="media">
  <div class="media-left">
    <a href="#">
      
    </a>
  </div>
  <div class="media-body">
    <h4 class="media-heading">
      {{ data.productName }}
    </h4>
    {{ data.releasedDate }}
    <rating
      [rating-value]="data.rating"
      [numOfReviews]="data.numOfReviews">
    </rating>
    <div [ngSwitch]="data.rating">
      <div *ngSwitchCase="1">Poor</div>
      <div *ngSwitchCase="2">Fair</div>
      <div *ngSwitchCase="3">Good</div>
      <div *ngSwitchCase="4">Very Good</div>
      <div *ngSwitchCase="5">Excellent</div>
      <div *ngSwitchDefault>Not Rated</div>
    </div>
  </div>
</div>
```

```

        {{ data.description }}
      </div>
    </div>
  `
,

```

So depending on the value of `data.rating`, we display different product comments. We use `*ngSwitchDefault` in the case where the value of `data.rating` does not match any of the cases, we display a default message ‘Not Rated’.

5.3 Pipes

To format data in Angular, we can use ‘pipes’. A pipe takes in data as input and transforms it to a desired output. Angular has built in pipes like `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. We can also create custom pipes which we will visit in the next section. Because pipes are relatively straightforward, we will illustrate the `DatePipe` and you can go on to explore the rest of the built in pipes on your own.

First, we change the value of `releasedDate` in the object returned by `ProductService` to a Javascript `Date` object as shown in bold below.

```

return [
  {
    imageUrl: "http://loremflickr.com/150/150?random=1",
    productName: "Product 1",
    releasedDate: new Date(2016,5,30),
    description: "...",
    rating: 4,
    numOfReviews: 2
  },

```

We have assigned `releasedDate` with a `Date` object with value 30 Jun, 2016. (Note that the month count starts from 0). Now, the date gets displayed as “Thu Jun 30 2016 00:00:00 GMT+0800 (SGT)” which is not what we want to display to the user. To use pipe formatting, in `product.component.ts`, we do the below,

```

{{ data.releasedDate | date }}

```

The date now gets displayed as Jun 30, 2016.

Parameterizing a Pipe

We can also supply pipes with optional parameters to fine tune its format. We do so by adding a colon (:) and then the parameter value (e.g., `date:"MM/dd/yy"`). The below

code returns our date 06/30/16 in the specified format of MM/dd/yy.

```
{{ data.releasedDate | date:"MM/dd/yy" }} // 06/30/16
```

Chaining Pipes

We can also chain pipes together for useful combinations. In the below example, we chain `releasedDate` to `DatePipe` and on to `UpperCasePipe` to display the date in uppercase. The following birthday displays as JUN 30, 2016 .

```
{{ data.releasedDate | date | uppercase }} // JUN 30, 2016
```

5.4 Custom Pipes

We can also write our own custom pipes. We use custom pipes the same way we use built-in pipes. We will implement a custom pipe that takes a string and truncate it to a specified length. This is useful for displaying a truncated product description in the view if it is too long like in figure 5.4.1.

Products

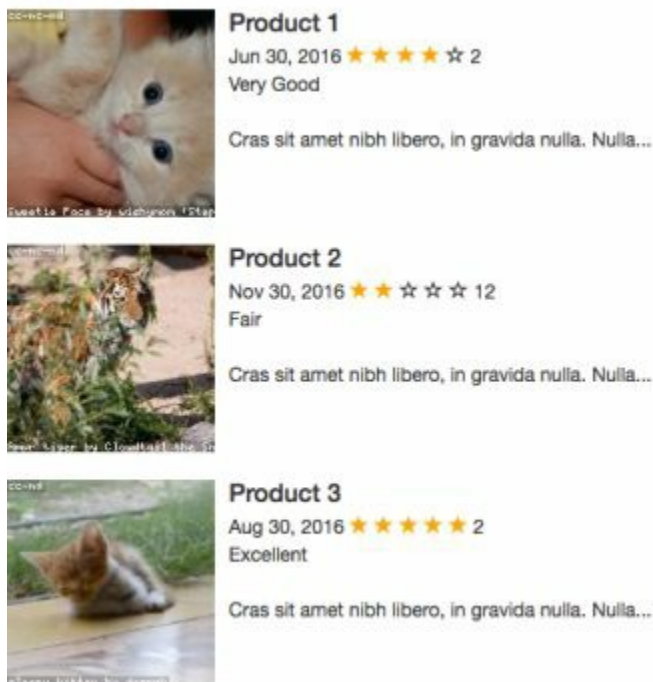


fig. 5.4.1

First, create a new file `truncate.pipe.ts` in the `app` folder. Type in the below codes into the file.

```
import { Pipe, PipeTransform } from '@angular/core';
```

```
@Pipe({name: 'truncate'})
```

```
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit:number): string{
    return value.substring(0,limit) + "...";
  }
}
```

Code Explanation

A pipe is a plain TypeScript class decorated with pipe metadata `@Pipe({name: 'truncate'})`. We tell Angular this is a pipe by applying the `@Pipe` decorator which we import from the core Angular library. The `@Pipe` decorator allows us to define the pipe name that we'll use within template expressions. Our pipe's name is `truncate`.

Every pipe class implements the `PipeTransform` interface's `transform` method. The `transform` method accepts an input value followed by optional parameters and returns the transformed value. In our code, `value` is the string we want to truncate and `limit` determines the limit of our truncation.

```
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit:number): string{
    return value.substring(0,limit) + "...";
  }
}
```

Final Steps

Next, we have to include our pipe in the `declarations` array of the `AppModule` to specify that our pipe is part of the `AppModule`. Add the lines below in **bold** to `app.module.ts`.

```
import { BrowserModule } from '@angular/platform-browser';
...
...
import { RatingComponent } from './rating.component';
```

```
import { TruncatePipe } from './truncate.pipe';
```

```
@NgModule({
  declarations: [
    AppComponent, ProductsComponent, RatingComponent, ProductComponent, TruncatePipe
  ],
  imports: [
    ...
  ],
  providers: [],
```

```
bootstrap: [AppComponent]
}))
export class AppModule { }
```

Finally in `product.component.ts` , add the pipe to the `data.description` interpolation like,
`{{ data.description | truncate: 20 }}`

5.6 ng-content

Sometimes, we need to insert content into our component from the outside. For example, we want to implement a component that wraps a bootstrap jumbotron. A bootstrap jumbotron (fig. 5.6.1) as defined on getbootstrap.com is “A lightweight, flexible component that can optionally extend the entire viewport to showcase key content on your site.”



fig. 5.5.1

Here is an implementation of the bootstrap jumbotron component.

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'bs-jumbotron',
  template: `
    <div class="jumbotron">
      <p></p>
      <p><a class="btn btn-primary btn-lg" href="#" role="button"></a></p>
    </div>
  `,
})
export class JumboTronComponent {
}
```

The markup above can be obtained from
<http://getbootstrap.com/components/#jumbotron>.

The selector is `bs-jumbotron` . We add a prefix `bs-` (‘bs’ as an abbreviation for bootstrap) to distinguish this component from other components that potentially might

have the same name. The jumbotron component is called in app.component.ts using,

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <bs-jumbotron></bs-jumbotron>
  `,
})
export class AppComponent {
}
```

To supply content to the jumbotron component, we can use Input properties. We can define an Input property in our jumbotron component and use property binding as shown below:

```
<bs-jumbotron [body]="..."></bs-jumbotron>
```

This is not ideal however. For we probably want to write a lengthier html markup here like,

```
<bs-jumbotron>
```

This is a simple hero unit, a simple jumbotron-style component for calling extra attention to featured content or information.

```
</bs-jumbotron>
```

We want to insert content into the jumbotron component from the outside. To do so, we define insertion points with ng-content as shown below into our jumbotron component template.

```
@Component({
  selector: 'bs-jumbotron',
  template: `
    <div class="jumbotron">
      <p><ng-content></ng-content></p>
      <p><a class="btn btn-primary btn-lg" href="#" role="button"></a></p>
    </div>
  `,
})
export class JumboTronComponent {
}
```



This is a simple hero unit, a simple jumbotron-style component for calling extra attention to featured content or information.

Multiple Insertion Points

We can also define multiple insertion points by adding the select directive. In `bs-jumbotron.component.ts`, we do this by using `ng-content` and specifying a css class to the `select` attribute to distinguish where content should go.

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'bs-jumbotron',
  template: `
    <div class="jumbotron">
      <h1><ng-content select=".heading"></ng-content></h1>
      <p><ng-content select=".body"></ng-content></p>
      <p><a class="btn btn-primary btn-lg" href="#" role="button">
        <ng-content select=".button"></ng-content></a></p>
    </div>
  `,
})
export class JumboTronComponent {
}
```

In the above code, we specify that content with the class `heading` will be placed in the `h1` heading tag. Content with class `body` will be placed in the body and content with class `button` will be in the button.

We now add in `app.component.ts` the rest of the text with the various css classes as shown below.

```
@Component({
  selector: 'app-root',
  template: `
    <bs-jumbotron>
      <div class="heading">
        Hello World!
      </div>
      <div class="body">
        This is a simple hero unit, a simple jumbotron-style component for calling extra
        attention to featured content or information.
      </div>
      <div class="button">
        Learn more
      </div>
    </bs-jumbotron>
  `,
})
```

```
})  
export class AppComponent {  
}
```

If you run your app now, you should now see something like in figure 5.6.2.



fig. 5.6.2

Summary

In this chapter, we introduced built in directives like `ngFor` , `ngIf` and `ngSwitch` that gives us more control in rendering our html. We learnt about formatting data using built in pipes like `DatePipe` , `UpperCasePipe` and creating our own custom pipes for custom formatting. We have also learnt about inserting content into components from the outside using `ng-content` .

Contact me at support@i-ducate.com if you encounter any issues or if you had not requested the full source code for this chapter.

CHAPTER 6: TEMPLATE DRIVEN FORMS

In Angular 2, we have template driven and model driven forms. Template driven forms are easier to implement and involve writing less code. But they give us limited control over validation. Model driven forms have more code but we have full control over validation. In this chapter, we will look at template driven forms.

6. 1 Create the User Model Class

Before we create our form, we want to define a class or model that can hold the data that we receive from the form. A model is represented by a simple class that contains properties. We will create a `User` class with two required fields (first name, email) and one optional field (country).

Create a new file in the app folder called `user.ts` with the following code:

```
export class Hero {  
  constructor(  
    public firstName: string,  
    public email: string,  
    public country?: string  
  ) { }  
}
```

Note that the `?` in `country` specifies that it's an optional field.

Next, create a new file called `user-form.component.ts` and give it the following definition:

```
import { Component } from '@angular/core';  
import { User } from './user';  
  
@Component({  
  selector: 'user-form',  
  templateUrl: 'user-form.component.html'  
})  
export class UserFormComponent {  
  countries = ['United States', 'Singapore',  
    'Hong Kong', 'Australia'];  
  
  model = new User("", "");
```

```

submitted = false;

onSubmit() {
  this.submitted = true;
}
}

```

In the `UserFormComponent`, we import the `User` model we just created. We specify our selector to be `user-form`. Because the html markup will be quite long, we use the `templateUrl` property to point to a separate html file `user-form.component.html` for our template.

6.2 Revising `app.module.ts`

Because template-driven form features are in `FormsModule`, we need to add it to the imports array in `app.module.ts` if we have not already done so. `FormsModule` gives our application access to template-driven forms features, including `ngModel`.

We also import the `UserFormComponent` and add it to declarations array in `app.module.ts` so that `UserFormComponent` is accessible throughout this module.

**Note:* I was initially confused on whether a component should be in the imports or declarations array. The simple rule is, if a component, directive, or pipe belongs to a module in the imports array, don't declare it in the declarations array. If you wrote a component and it should belong to this module, do declare it in the declarations array.

6.3 Create an initial HTML Form Template

Next, create our template file `user-form.component.html` and give it the following definition:

```

<div class="container">
  <h1>User Form</h1>
  <form>
    <div class="form-group">
      <label for="firstName">First Name</label>
      <input type="text" class="form-control" id="firstName" required>
    </div>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="text" class="form-control" id="email" required>
    </div>
    <div class="form-group">
      <label for="country">Country</label>

```

```

    <input type="text" class="form-control" id="country">
    <select class="form-control" id="country">
      <option *ngFor="let c of countries" [value]="c">{{c}}</option>
    </select>
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
</div>

```

Our form definition till now is just plain html5. We are not using Angular yet. We allow firstName and email fields for user input in input boxes. Both firstName and email has the html5 required attribute; the country does not have required because it is optional.

At the end we have a Submit button.

6.4 Using *ngFor to Display Options

In the country field, we choose one country from a fixed list of countries using select dropdown. We will retrieve the list of countries from an array in UserFormComponent and bind the options to the country list using ngFor with the below code.

```

<div class="form-group">
  <label for="country">Country</label>
  <select class="form-control" id="country">
    <option *ngFor="let c of countries" [value]="c">{{c}}</option>
  </select>
</div>

```

With *ngFor , we repeat the <option> tag for each country in the list of countries. The variable c holds a different country element in each iteration and we display it using interpolation {{c}} .

6.5 Two-way data binding with ngModel

We use [(ngModel)] to easily create two way binding between our form and the User model.

Now, update <input> for firstName like this

```

<div class="form-group">
  <label for="firstName">First Name</label>
  <input type="text" class="form-control" id="firstName" required

```

```
        [(ngModel)]="model.firstName" name="firstName">
    </div>
```

We have also added a name attribute to our `<input>` tag and set it to `firstName`. We can assign any unique value, but we should use a descriptive name like the name of the input. The name attribute is required by Angular Forms to register the control with the form.

Do the same for the rest of the fields as in below.

```
<div class="container">
  <h1>User Form</h1>
  <form>
    <div class="form-group">
      <label for="firstName">First Name</label>
      <input type="text" class="form-control" id="firstName" required
        [(ngModel)]="model.firstName" name="firstName">
    </div>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="text" class="form-control" id="email" required
        [(ngModel)]="model.email" name="email">
    </div>
    <div class="form-group">
      <label for="country">Country</label>
      <select class="form-control" id="country"
        [(ngModel)]="model.country" name="country">
        <option *ngFor="let c of countries" [value]="c">{{c}}</option>
      </select>
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
</div>
```

Note that the `id` property in each input or select is used by the label to match to its `for` attribute so that when a user clicks on the label, the input or select receives the focus therefore enabling better user interaction since a user can either click on the label or input itself for the right focus.

If we run the application now, we should get something like in figure 6.5.1.

User Form

First Name

Greg

Email

greg@gmail.com

Country

Singapore

Submit

figure 6.5.1

6.6 Track change-state and validity with ngModel

ngModel not only provide us with two way data binding between the form and controls, it also allows us to track change state and validity of the controls. This is useful to let us know if an input has been filled in and if the input is valid so that we can display alert or warning messages to the user.

For a control with ngModel directive, if a user has touched the control, the control's Angular css class ng-touched returns true and ng-untouched returns false. The reverse is true.

If the value of the control is changed, its ng-dirty class returns true and ng-pristine returns false. The reverse is true.

If the value of the control is valid, ng-valid class returns true and ng-invalid returns false. The reverse is true.

In the following, we illustrate how to make use of these css classes to display alert or warning messages when an input is improperly filled in.

Show and Hide Validation Error messages

In the below code, we firstly add the #firstName variable and give it the value ngModel to create a reference to the firstName input control so that we can access its css classes. Next, we show the "First Name is required" message if firstName.touched is true and firstName.valid is false. Note that we have divided the markup of the input element into multiple lines instead of one single line so that it is easier to see its attributes.

```
<div class="form-group">
  <label for="firstName">First Name</label>
  <input type="text"
    class="form-control"
    id="firstName" required
```

```

    [(ngModel)]="model.firstName"
    name="firstName"
    #firstName="ngModel"
  >
  <div class="alert alert-danger"
    *ngIf="firstName.touched && !firstName.valid ">
    First name is required
  </div>
</div>

```

Why do we check touched ? This is to avoid showing errors before the user has had a chance to edit the value, for example when the form is freshly loaded. This prevents premature display of errors.

We proceed to do the same for email input.

```

<div class="form-group">
  <label for="email">Email</label>
  <input type="text"
    class="form-control"
    id="email" required
    [(ngModel)]="model.email"
    name="email"
    #email="ngModel">
  <div *ngIf="email.touched && !email.valid"
    class="alert alert-danger">Email is required.
  </div>
</div>

```

We don't show or hide any validation messages for country since it is optional so we leave it as it is.

6.7 Showing Specific Validation Errors

Other than the required validation, we can also specify our input validation on an element's minimum and maximum length using its minlength and maxlength html validation attributes. The below code illustrates this.

```

<div *ngIf="firstName.touched && firstName.errors">
  <div class="alert alert-danger"
    *ngIf="firstName.errors.required">
    First name is required
  </div>
  <div class="alert alert-danger"
    *ngIf="firstName.errors.minlength">
    First name should be minimum 3 characters.
  </div>
</div>

```

```
</div>
</div>
```

The first `*ngIf` on the outer `<div>` element reveals two nested message divs if the control is touched and there are `firstName` errors.

Inside the outer `<div>` are two nested `<div>` which presents a custom message for required and minlength validation error.

6.8 Submit the form with `ngSubmit`

To implement submitting of forms, we update the `<form>` tag with the Angular directive `NgSubmit` and bind it to the `UserFormComponent.submit()` method with an event binding in the below code.

```
<form (ngSubmit)="onSubmit()" #userForm="ngForm">
```

In the `<form>` tag, we also define a variable `#userForm` and assign the value “`ngForm`” to it to make `userForm` a reference to the form as a whole. The `ngForm` directive allows `userForm` to hold control to its contained elements with the `ngModel` directive and name attribute. `userForm` thus allows us to monitor these contained control elements’ properties including their validity. `userForm` also has its own `valid` property which is true only if every contained control is valid. This enables us to bind the Submit button's `disabled` property to the form's over-all validity via the `#userForm` variable. That is, the Submit button remains disabled until all required inputs are filled up properly. We do this with the below code in the Submit button.

```
<button type="submit" class="btn btn-default"
    [disabled]="!userForm.form.valid">Submit</button>
```

Now when either `firstName` or email field is not filled up, the error is shown and the Submit button is disabled.

6.9 Getting Submitted Values

To show that we have successfully submitted our form with its values, add the below code to the bottom of `user-form.component.html`.

```
<div [hidden]="!submitted">
    You submitted the following:
    First Name: {{ model.firstName }}<br>
    Email: {{ model.email }}<br>
    Country: {{ model.country }}<br>
    <button class="btn btn-default"
```

```
(click)="submitted=false">Remove</button>
</div>
```

When we click submit, we display the values entered with interpolation bindings (fig. 6.9.1). The values only appears while the component is in the submitted state. We added a Remove button whose click event sets submitted to false which removes the values.



fig. 6.9.1

Here is the final code of user-form.component.html

```
<div class="container">
  <h1>User Form</h1>
  <form (ngSubmit)="onSubmit()" #userForm="ngForm">
    <div class="form-group">
      <label for="firstName">First Name</label>
      <input type="text"
        class="form-control"
        id="firstName" required
        [(ngModel)]="model.firstName"
        name="firstName"
        #firstName="ngModel"
        minlength="3">
      <!-- display validation error-->
      <div *ngIf="firstName.touched && firstName.errors">
        <div class="alert alert-danger"
          *ngIf="firstName.errors.required">
          First name is required
        </div>
        <div class="alert alert-danger"
          *ngIf="firstName.errors.minlength">
          First name should be minimum 3 characters.
        </div>
      </div>
    </div>
  </form>
</div>
```



```

</div>
<div class="form-group">
  <label for="email">Email</label>
  <input type="text"
    class="form-control"
    id="email" required
    [(ngModel)]="model.email"
    name="email"
    #email="ngModel"
    minlength="3">
  <!-- display validation error-->
  <div *ngIf="email.touched && email.errors">
    <div class="alert alert-danger"
      *ngIf="email.errors.required">
      Email is required
    </div>
    <div class="alert alert-danger"
      *ngIf="email.errors.minlength">
      Email should be minimum 3 characters.
    </div>
  </div>
</div>
<div class="form-group">
  <label for="country">Country</label>
  <select class="form-control" id="country"
    [(ngModel)]="model.country" name="country">
    <option *ngFor="let c of countries"
      [value]="c">{{ c }}</option>
  </select>
</div>
<button type="submit" class="btn btn-default"
  [disabled]="!userForm.form.valid">Submit</button>
</form>
</div>
<div [hidden]="!submitted">
  You submitted the following:
  First Name: {{ model.firstName }}<br>
  Email: {{ model.email }}<br>
  Country: {{ model.country }}<br>
  <button class="btn btn-default" (click)="submitted=false">Remove</button>
</div>

```

Here is the final code of user-form.component.ts

```

import { Component } from '@angular/core';
import { User } from '../user';

```

```

@Component({

```

```

    selector: 'user-form',
    templateUrl: 'user-form.component.html'
  })
  export class UserFormComponent {
    countries = ['United States', 'Singapore',
      'Hong Kong', 'Australia'];

    model = new User("", "", "");

    submitted = false;

    onSubmit() {
      this.submitted = true;
    }
  }
}

```

Summary

In this chapter, we learnt how to create a template driven form. We created a model class to represent the data in the form, used `*ngFor` to display the list of options in a select dropdown, created two-way binding with form and component properties using `ngModel`, used `ngModel` to help us show form field validation errors, and finally learnt form submission using `ngSubmit`.

CHAPTER 7: MODEL DRIVEN FORMS

In the previous chapter, we learn about template driven forms. With template driven forms however, we are limited to a few basic validators. If we want to implement custom validation, we need to use model driven forms which we will cover in this chapter.

7.1 Building a Basic Login Form

We will first create a component that renders a simple login form. In app, create a new file `login.component.ts` with the below code.

```

import { Component } from '@angular/core';

@Component({
  selector: 'login',
  templateUrl: 'login.component.html'
})

```

```

}))
export class LoginComponent {

}

```

Next create login.component.html as referenced in templateUrl property in login.component.ts . Fill in login.component.html with the below code.

```

<form>
  <div class="form-group">
    <label for="username">Username</label>
    <input id="username" type="text" class="form-control">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input id="password" type="password" class="form-control">
  </div>
  <button class="btn btn-primary" type="submit">Login</button>
</form>

```

The template above is currently just pure html with a few bootstrap classes. It does not have any validation or any Angular directives yet. In the next few sections, we will upgrade our form and implement custom validation on the username and password field.

7.2 Creating Controls Explicitly

To upgrade our form into an Angular form that can perform validation, we have to create FormControl and FormGroup objects. A FormControl object represents a form control in a form. With it, we can track the value and validation status of an individual form control. A FormGroup object tracks the value and validity state of a *group* of FormControl objects.

In login.component.ts , add the codes in **bold** shown below.

```

import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector:'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {
  form = new FormGroup();
}

```

We have created a property called form and initialize it with new FormGroup(). FormGroup and FormControl are imported from angular/forms .

We next pass in two FormControl objects, username and password into FormGroup() . as show below.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector:'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {
  form = new FormGroup({
    username: new FormControl ('', Validators.required),
    password: new FormControl ('', Validators.required)
  });
}
```

The first parameter in the FormControl constructor is optional. It is the default value we give to the control. This is useful when we display existing data in the control for user to edit. We first retrieve the existing value from the server and populate the control with it.

The second parameter takes in a Validator function. Validators is defined in angular/commons and provides a set of validators like required , minlength and maxlength . For now, we have used Validators.required .

Back in the template, we need to tell Angular that we have created a FormGroup and its FormControls explicitly so that Angular will not create them for us as in the case of template driven forms.

In login.component.html , add the codes in **bold**.

```
<form [formGroup]="form">
  <div class="form-group">
    <label for="username">Username</label>
    <input id="username" type="text" class="form-control"
      formControlName="username">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input id="username" type="password" class="form-control"
      formControlName="password">
  </div>
  <button class="btn btn-primary" type="submit">Login</button>
```

```
</form>
```

Code Explanation

```
<form [formGroup]="form">
```

In the form element, we apply the `formGroup` directive and bind it to “form”.

```
    <input id="username" type="text" class="form-control"
          formControlName="username">
```

We then associate each input field to the `FormControl` object by using `formControlName` directive `formControlName="username"`. This is important for referencing our `FormControl` object from `FormGroup`. Angular will look at the `FormGroup` object and expects to find the `FormControl` object with the exact name we specify in `formControlName="username"`. If it can't find the control with that name, it will throw an exception.

Implementing Validation

Next, we define validation message placeholders that will be displayed when the input is invalid. Add the following codes in **bold**.

```
<form [formGroup]="form">
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" class="form-control"
          formControlName="username">
    <div *ngIf="form.controls.username.touched &&
              !form.controls.username.valid" class="alert alert-danger">
      Username is required
    </div>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control"
          formControlName="password">
    <div *ngIf="form.controls.password.touched &&
              !form.controls.password.valid" class="alert alert-danger">
      Password is required
    </div>
  </div>
  <button class="btn btn-primary" type="submit">Login</button>
</form>
```

Code Explanation

We check username css classes touched and valid to see if the control has been touched and input value is not valid. If so, display the div validation placeholder message “Username is required” with alert and alert-danger bootstrap classes. We do the same for password (figure 7.2.1).

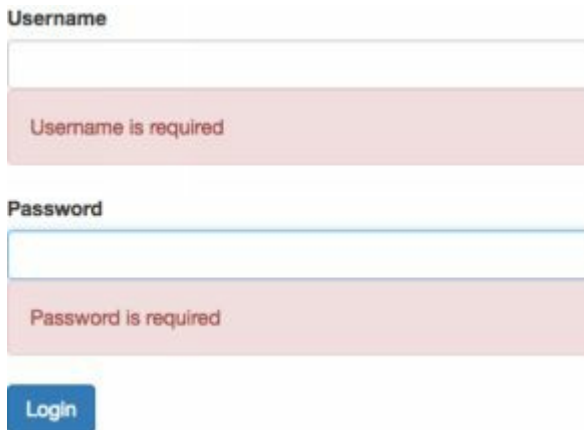
The image shows a login form with two input fields. The first field is labeled 'Username' and the second is labeled 'Password'. Both fields have a red border and a red background, indicating they are required and currently empty. Below each field is a red message box with the text 'Username is required' and 'Password is required' respectively. At the bottom of the form is a blue button labeled 'Login'.

figure 7.2.1

Submitting the Form

To handle the submit event of the form, bind the `ngSubmit` event to the `login()` method in the form element as shown below.

```
<form [formGroup]="form" (ngSubmit)="login()">
```

Now in `login.component.ts` , we implement the `login` method.

```
export class LoginComponent {  
  form = new FormGroup({  
    username: new FormControl ('', Validators.required),  
    password: new FormControl ('', Validators.required)  
  });  
  
  login(){  
    console.log(this.form.value); // prints form values in json format  
  }  
}
```

`login()` currently just prints the submitted values in json format.

app.component.ts

Next in `app.component.ts` , render the `login` component in the template:

```
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  template: `
    <login></login>
  `
})
export class AppComponent {
}
```

Lastly in `app.module.ts` , import `ReactiveFormsModule` and add it to the imports array as shown below. `ReactiveFormsModule` provides the classes for implementing model driven forms.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
```

```
import { AppComponent } from './app.component';
import { LoginComponent } from './login.component';
```

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

What we have covered so far for model driven forms (e.g. Validation) is similar to template driven forms. The difference is that in model driven forms we are explicitly creating `FormGroup` and `FormControl` objects explicitly (in template driven forms,

Angular creates it for you), and we tell Angular about it in the template by using the `formGroup` and `formControlName` directive.

7.3 Using FormBuilder

We can use the `FormBuilder` class to declare `FormControl` and `FormGroup` objects in a more compact way. For example, the previous form can also be implemented with `FormBuilder` as shown below.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector:'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {

  form: FormGroup;

  constructor(fb: FormBuilder){
    this.form = fb.group({
      username:['',Validators.required],
      password:['',Validators.required]
    })
  }

  login(){
    console.log(this.form.value);
  }
}
```

Code Explanation

```
import { FormBuilder, FormGroup, FormControl, Validators } from '@angular/forms';
```

We first import the `FormBuilder` class from `@angular/forms` . Because we are no longer using `FormControl` class, we can remove it from the import.

```
form: FormGroup;

constructor(fb: FormBuilder){
  this.form = fb.group( {
    username:['',Validators.required],
    password:['',Validators.required]
  })
}
```



```
}
```

In the constructor, we use dependency injection to get an instance of FormBuilder, fb. FormBuilder has a method group which takes in a shortened code for new FormControl as compared to:

```
form = new FormGroup({  
  username: new FormControl('', Validators.required),  
  password: new FormControl('', Validators.required)  
});
```

Essentially, FormBuilder is syntactic sugar that shortens the new FormGroup(), new FormControl() code that can build up in larger forms. This results in cleaner and more compact for large forms.

```
this.form = fb.group({  
  username:['',Validators.required],  
  password:['',Validators.required]  
})
```

Like new FormControl, the first element is the default value and the second element is the validator function. Note that both are optional and it is not mandatory to specify them.

Finally, the group method returns a FormGroup object as a result and we store it in form.

7.4 Implementing Custom Validation

While there are a couple of built-in validators provided by Angular like required, min/maxlength, we often need to add some custom validation capabilities to our application's form to fulfill our needs for example, a valid password cannot contain a space. In this section, we will implement a custom validator that checks if a password has space in it.

First, in app, add a new file passwordValidator.ts. This class will include all validation rules for the password field. Fill it in with the below codes.

```
import { FormControl } from '@angular/forms';  
  
export class PasswordValidator{  
  static cannotContainSpace(formControl: FormControl){  
    if(formControl.value.indexOf(' ') >= 0)  
      return { cannotContainSpace: true };  
  }  
}
```

```

    return null;
  }
}

```

Code Explanation

We declare a static method `cannotContainSpace` that takes in a `FormControl` object as argument. We access the `value` string property of the `formControl` and check if there are spaces in it with the `indexOf` method. If there are, we return `{ cannotContainSpace: true }` and if there are not spaces, we return `null`.

Note that Angular validators work this way; If the validation passes, return `null`. If it fails, return `{<validationRule>:<value>}` where `<value>` can be anything. It can be a boolean `true/false`, or an object that supplies more data about the validation error.

login.component.ts

Next, we apply the `cannotContainSpace` validator to the password field. Back in `login.component.ts`, add the lines in **bold**.

```

import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

```

```

import { PasswordValidator } from './passwordValidator';

```

```

@Component({
  selector:'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {

```

```

  form: FormGroup;

```

```

  constructor(fb: FormBuilder){
    this.form = fb.group({
      username:['',Validators.required],
      password:['',Validators.compose([Validators.required,
        PasswordValidator.cannotContainSpace])]
    })
  }

```

```

  login(){
    console.log(this.form.value);
  }
}

```

Code Explanation

First, we import our PasswordValidator , and apply it to password . Because we have more than one validator on password form control, we need to compose the multiple validators by calling Validators compose method. The compose method takes in an array of Validators

[Validators.required, PasswordValidator.cannotContainSpace]

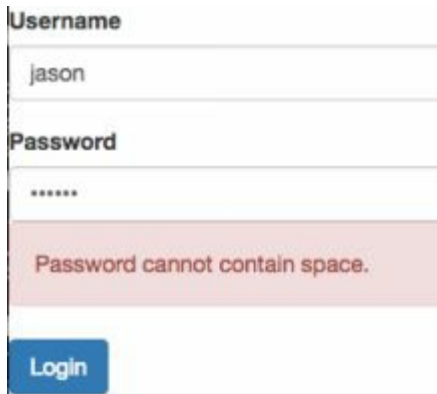
login.component.html

Finally, we implement the custom validation message placeholder. Back in login.component.html , replace the validation codes for password as shown in **bold** below.

```
<form [formGroup]="form" (ngSubmit)="login()">
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" class="form-control" formControlName="username">
    <div *ngIf="form.controls.username.touched
      && !form.controls.username.valid" class="alert alert-danger">
      Username is required
    </div>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" formControlName="password">
    <div *ngIf="form.controls.password.touched
      && form.controls.password.errors">
      <div *ngIf="form.controls.password.errors.required"
        class="alert alert-danger">
        Password is required.
      </div>
      <div *ngIf="form.controls.password.errors.cannotContainSpace"
        class="alert alert-danger">
        Password cannot contain space.
      </div>
    </div>
  </div>
  <button class="btn btn-primary" type="submit">Login</button>
</form>
```

Notice that we instead of checking for !password.valid , we now check for password.errors . This is because there are now more than one kinds of error for password namely, required and cannotContainSpace , so we need to further have a nested *ngIf for each error to check for each kind of error specifically.

When you run your app now and try to enter a password with space in it, you will get a validation error message like in figure 7.4.1.



The image shows a web form with two input fields. The first field is labeled 'Username' and contains the text 'jason'. The second field is labeled 'Password' and contains several dots, indicating a masked password. Below the password field, there is a red rectangular box containing the text 'Password cannot contain space.' in a red font. At the bottom of the form, there is a blue button with the text 'Login' in white.

figure 7.4.1

7.5 Validating Upon Form Submit

There are times when you have to do validation upon submitting the form to the server. For example, validating username and password against the application's database. In the below example, we will illustrate validation of username and password upon submitting the login form.

First in app folder, create a new file login.service.ts which is the service class for login functionality. Remember that we should implement logic in service classes to keep our component classes lightweight and mainly for rendering displays. Fill in login.service.ts with the below code.

```
import {Injectable} from '@angular/core';

@Injectable()
export class LoginService {
  login(username, password){
    if(username === "jason" && password === "123")
      return true;
    else
      return false;
  }
}
```

Our login service class is a simple class with a method login that takes in argument username and password credentials. We mark our Login service class as available for dependency injection by decorating it with the @Injectable() annotation.

In a real application, our login method should call an authentication api on a server with the credentials. To simplify our illustration for now, we authenticate with

hardcoded values.

Do note that whenever we create a new service class and want to use it, we should specify it in the providers array of our module class to state that we want to use this service in that module. In our case, since we have only one module AppModule , we specify it in providers: [LoginService] of app.module.ts . We will cover more about modules and their providers in **Chapter 11 - Structuring Large Apps With Modules**.

Next in login.component.ts , add the below codes in **bold**.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { PasswordValidator } from './passwordValidator';
import { LoginService } from './login.service';

@Component({
  selector:'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent {

  form: FormGroup;

  constructor(fb: FormBuilder, private _loginService: LoginService){

    this.form = fb.group({
      username:["",Validators.required ],
      password:["",Validators.compose([Validators.required,
        PasswordValidator.cannotContainSpace])]
    })
  }

  login(){
    var result = this._loginService.login(this.form.controls['username'].value,
      this.form.controls['password'].value);

    if(!result){
      this.form.controls['password'].setErrors({
        invalidLogin: true
      });
    }
  }
}
```

Code Explanation

We import and use dependency injection in the constructor to get an instance

of `LoginService`. We then have a method `login()` that calls the login method in our `loginService` instance with the user-keyed in values of username and password.

```
this.form.controls['username'].value  
this.form.controls['password'].value
```

With the above code, we access the value property of username and password control inside form to get the user keyed-in values.

```
if(!result){  
  this.form.controls['password'].setErrors({  
    invalidLogin: true  
  });  
}
```

The login method returns result as true if the login credentials are valid. If false, we access the password FormControl with `this.form.controls['password']` and call its `setErrors` method to supply the error `invalidLogin: true`. As mentioned earlier, true can also be replaced with a value or object to provide more details about the validation.

login.component.html

Lastly, we add a div to `login.component.html` for our login validation message.

In `login.component.html`, add the below codes in **bold**.

```
<div class="form-group">  
  <label for="password">Password</label>  
  <input type="password" class="form-control" formControlName="password">  
  <div *ngIf="form.controls.password.touched && form.controls.password.errors">  
    <div *ngIf="form.controls.password.errors.invalidLogin"  
      class="alert alert-danger">  
      Username or password is invalid.  
    </div>  
    <div *ngIf="form.controls.password.errors.required"  
      class="alert alert-danger">  
      Password is required.  
    </div>  
    <div *ngIf="form.controls.password.errors.cannotContainSpace"  
      class="alert alert-danger">  
      Password cannot contain space.  
    </div>  
  </div>  
</div>
```

We use a `*ngIf` to check for the error `invalidLogin` and display the alert message “Username or password is invalid”. Run your app now and if you do not supply a

valid username and password , you should get the invalid login message as shown in figure 7.5.1.



Username

sad

Password

.....

Username or password is invalid.

Login

figure 7.5.1

Summary

In this chapter, we learnt how to implement model driven forms in Angular. We learnt how to create FormControl and FormGroup objects, use FormBuilder to make our code more compact, how to implement custom validation, and how to validate the form upon submit. Now after submitting a form, we need to persist the data by calling the api endpoint of the server. We will begin to explore on how to communicate with the server in the next few chapters.

CHAPTER 8: INTRODUCTION TO OBSERVABLES

For Angular 2 to connect to backend servers, we need Observables which is a concept introduced in a library called Reactive extensions. Reactive Extensions is a comprehensive library by itself independent from Angular. If you wish to know more about it, you can go to reactivex.io. Reactive Extensions have been implemented for Java, Javascript, .Net, Scala, Clojure, Swift and other languages. Reactive Extensions is used in a few parts of Angular 2 especially in the classes we use to connect to a server. The rxjs folder in node_modules contains the Reactive Extensions library for Javascript used by Angular 2.

8.1 Observables

Observables are at the core of Reactive Extensions. An Observable represents an asynchronous data stream where data arrives asynchronously. An example is the keyup event from a textbox. Another example is a web socket. A server uses web-sockets to send data to the client. Data arrives asynchronously and we can model it as an Observable stream.

We can think of Observables as a collection. The difference is that in a collection we iterate through the collection and pull each object one at a time as shown below

```
foreach(var obj in collection){  
    ...  
}
```

For an Observable stream, we subscribe to it and give it a **callback function**.

```
function(newData){  
    ...  
}
```

When a new data element arrives, it will push that data element to us by calling the provided function back. We thus call the provided function a **callback function**.

Because we observe Observable streams and get notified when data arrives asynchronously, we call them Observables.

8.2 Creating an observable from DOM events

We will illustrate Observables by creating an Observable stream from the keyup event of a textbox. Suppose we want to implement a search service to look up artists' data on spotify.com. We do this by typing a search term into the input box and then call the spotify api to get artists data with a similar name.

First, either re-use the project you have from previous chapters or create a new project using Angular CLI. Then, copy the following code into app component.

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <input class="form-control" type="search"
      [formControl]="searchControl">
  `
})
export class AppComponent {
  searchControl = new FormControl();

  constructor(){
    this.searchControl.valueChanges.subscribe(value => {
      console.log(value);
    });
  }
}
```

Code Explanation

```
<input class="form-control" type="search"
  [formControl]="searchControl">
```

Because our search input consists of just one input control, we do not need a `<form>` element for it. We can use the `formControl` directive in the input control. The `formControl` represents our input field.

The `formControl` class has a property `valueChanges` (see below) which returns an Observable. We can subscribe to this observable by calling the `subscribe` method. In this way, we get notified whenever the value in the input field changes.

```
constructor() {
  this.searchControl.valueChanges.subscribe(value => {
    console.log(value);
  });
}
```

The `subscribe` method requires a function as argument. This function will be called by the Observable when new data arrives (i.e. value in input field changes). You can declare a function using the traditional anonymous function syntax like

```
function (value){  
  console.log(value);  
}
```

or you can shorten this code and use the arrow function (or lambda expression syntax).

```
value => {  
  console.log(value);  
}
```

Every time we press a key in the textbox, we get the text value pushed to us in an Observable stream. Our callback function gets called and we print the value to the console (fig. 7.2.1).

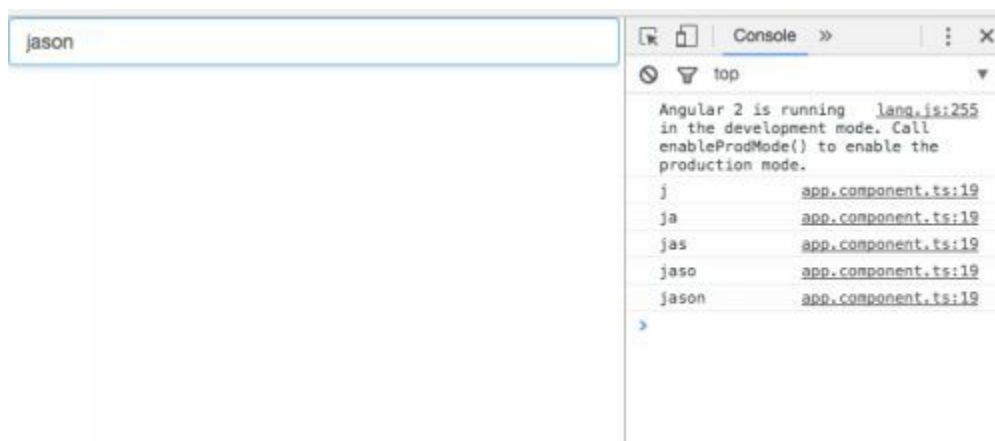


figure 8.2.1

8.3 Observable Operators

So what are the benefit of Observables? The benefit of Observables are that it provides a set of operators that we can use to transform, filter, aggregate and combine data received from the observable stream. In the following, we look at some of the operators:

filter Operator

In our input, say we want to call Spotify only if the user types at least three characters so as not to flood Spotify with too many requests. To do so, we can apply the filter operator. We do so by first importing the filter operator with the below code

```
import 'rxjs/add/operator/filter';
```

and add the filter operator as shown below in **bold**.

```
this.searchControl.valueChanges
  .filter(text => text.length >= 3)
  .subscribe(value => {
    console.log(value);
  });
```

The filter operator takes an expression `text.length >= 3` and determines that the value should be selected only if the expression returns true. filter then returns another Observable that we can either subscribe to or apply another operator. If we run the app now, we see only text of length greater or equal to three logged in the console.

debounceTime Operator

Suppose we want to wait 400 milliseconds in between requests before calling Spotify, we can apply the debounceTime operator. We do so by first importing the debounceTime operator with the below code

```
import 'rxjs/add/operator/debounceTime';
```

Next, we add the debounceTime operator as shown below.

```
this.searchControl.valueChanges
  .filter(text => text.length >= 3)
  .debounceTime(400)
  .subscribe(value => {
    console.log(value);
  });
```

Like the filter operator, the debounceTime operator returns an Observable which can be subscribed to. Thus, you can see that the benefit of Observables is that you can keep applying operators for the custom logic that you want.

distinctUntilChanged Operator

Say if a user presses the left and right arrow keys to move the cursor, the valueChange event is fired and we send multiple requests with the same input string to Spotify since the text in the input field is not changed. To avoid such multiple requests with the same search term, we can apply the distinctUntilChanged operator which will let us receive our Observable only when the text is changed. We do so by first importing the distinctUntilChanged operator with the below code

```
import 'rxjs/add/operator/distinctUntilChanged';
```

Next, add the `distinctUntilChanged` operator as shown below.

```
this.searchControl.valueChanges
  .filter(text => text.length >= 3)
  .debounceTime(400)
  .distinctUntilChanged()
  .subscribe(value => {
    console.log(value);
  });
```

Final Code

Below shows the final code for our app component.

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Rx';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import { FormControl } from '@angular/forms';
```

```
@Component({
  selector: 'app-root',
  template: `
    <input class="form-control" type="search"
      [formControl]="searchControl">
  `,
})
export class AppComponent {
  searchControl = new FormControl();

  constructor(){
    this.searchControl.valueChanges
      .filter(text => text.length >= 3)
      .debounceTime(400)
      .distinctUntilChanged()
      .subscribe(value => {
        console.log(value);
      });
  }
}
```

Summary

In this chapter, we are introduced to Observables, how to subscribe to Observables from DOM events, and how to apply certain Observable operators

like `filter` , `debounceTime` and `distinctUntilChanged` to avoid sending multiple repeated requests.

Now that we can get an `Observable` stream from an input, we will learn how to use the search terms keyed into the input to get data from a server in the next chapter.

CHAPTER 9: CONNECTING TO SERVER

With the knowledge of how to subscribe to Observables, we will see how to call backend services to get data through RESTful APIs in this chapter.

9.1 A Simple RESTful API

Building RESTful APIs is beyond the scope of Angular because Angular is a client side technology whereas building RESTful APIs require server side technology like NodeJS, ASP.NET, Ruby on Rails and so on. (But later on in chapter 12, we will introduce Firebase, which provides us with a simple way for us to create and store server-side data that we can utilize to build a fully functioning Angular application!)

We will illustrate by connecting to the Spotify RESTful API to retrieve and manage spotify content. You can know more about the Spotify API at

<https://developer.spotify.com/web-api/user-guide/>.

But as a quick introduction, we can get Spotify data with the following url,

<https://api.spotify.com/v1/search?q=<search term>&type=artist>

We simply specify our search term in the url to get Spotify data for artists with name matching our search term. An example is shown below with search term jason .

<https://api.spotify.com/v1/search?q=jason&type=artist>

When we make a call to this url, we will get the following json objects as a result (fig. 9.1.1).

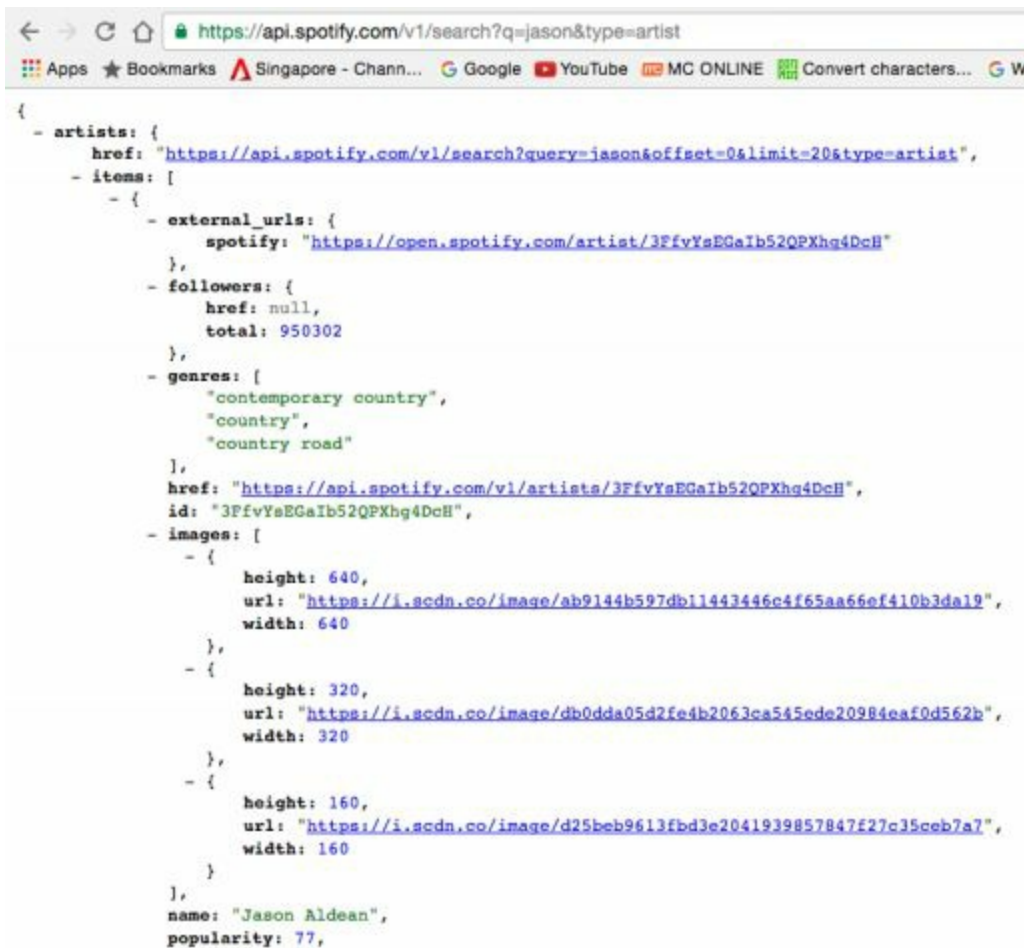


fig. 9.1.1

9.2 Getting Data from the Server

To get data using a RESTful API, we are going to use the `Http` class in Angular. We use it to make ajax calls to the server. The `Http` class provides the `get()` method for getting a resource, `post()` for creating it, `put()` for updating it, `delete()` for deleting resource and `head()` for getting metadata regarding a resource.

We will first create a service which will be responsible for talking to our RESTful API. Our components should not make http calls directly but rather delegate that role to services.

In the app folder, create a new file `spotify.service.ts` with the below code.

```

import {Http} from '@angular/http';
import 'rxjs/add/operator/map';

export class SpotifyService{
  constructor(private _http: Http){

  }

  getSpotifyData(){

```

```

        return this._http.get("https://api.spotify.com/v1/search?q=jason&type=artist")
            .map(res => res.json())
    }
}

```

getSpotifyData is a method that will return Spotify data from our api end point. To call our api end point, we need to use the Http service of Angular. We import it using

```
import {Http} from '@angular/http' .
```

We inject the Http class into the constructor of our Spotify Service. Remember dependency injection? We let Angular create an instance of Http class and give it to us. Our constructor has a parameter _http which is of type Http . By convention, we prefix private fields with an underscore ‘ _ ’.

```

constructor(private _http: Http){

}

```

In getSpotifyData , we use the get() method of http and give the url of our api endpoint. Note that we are currently hard-coding our search term to be jason . This will be changed later when we use the search term provided by the user from an input. The return type of get() is an Observable of <Response> . So we will return this Observable in our service and our component will be the consumer of this Observable. We will subscribe to it and when an ajax call is completed, the response is fed to the Observable and then pushed to the component.

```

getSpotifyData(){
    return this._http.get("https://api.spotify.com/v1/search?q=jason&type=artist")
        .map(res => res.json());
}

```

We use the map operator to transform our response object into json type.

```
.map(res => res.json())
```

The map operator has previously been imported with import 'rxjs/add/operator/map';

9.3 Dependency Injection

Next, we inject our service into our component. As covered in chapter 7, we do so by first marking our Spotify service class as available for dependency injection. We decorate it with the @Injectable() annotation as shown below.

```
import {Http} from '@angular/http';
```



```
import 'rxjs/add/operator/map';
import { Injectable } from '@angular/core';

@Injectable()
export class SpotifyService{
  constructor(private _http: Http){

  }

  getSpotifyData(){
    return this._http
      .get("https://api.spotify.com/v1/search?q=jason&type=artist")
      .map(res => res.json());
  }
}
```

Code Explanation

After importing the Injectable annotation.

```
import { Injectable } from '@angular/core';
```

We then apply it to the top of the class

```
@Injectable()
export class SpotifyService{
```

and the service is now ready for injection. Now let's go to app.component.ts .

app.component.ts

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Rx';

import { SpotifyService } from './spotify.service';

@Component({
  selector: 'app-root',
  template: `
    `,
  providers: [SpotifyService]
})
export class AppComponent {
  constructor(private _spotifyService: SpotifyService){
    this._spotifyService.getSpotifyData()
      .subscribe(data => console.log(data));
  }
}
```

```
}
```

Code Explanation

First, we import the service.

```
import { SpotifyService } from './spotify.service';
```

We then inject it into the constructor.

```
constructor(private _spotifyService: SpotifyService){  
  this._spotifyService.getSpotifyData()  
    .subscribe(data => console.log(data));  
}
```

Remember that in dependency injection, Angular looks at the constructor and sees a parameter of type `SpotifyService` and seeks to create an instance of it. To let the injector know where to find the `SpotifyService` to create it, we register the `SpotifyService` by specifying it in the `providers` property in the component's metadata.

```
@Component({  
  selector: 'app-root',  
  template: `  
    `,  
  providers: [SpotifyService]  
})
```

Now that we are done with the dependency injection, let's use our service. In the constructor, we call the `getSpotifyData` method from our `_spotifyService` instance. The `getSpotifyData` method returns an `Observable` which we need to subscribe to.

```
constructor(private _spotifyService: SpotifyService){  
  this._spotifyService.getSpotifyData()  
    .subscribe(data => console.log(data));  
}
```

We then pass in our callback function `data => console.log(data)`. Back in the implementation of `getSpotifyData` (see below), we use the `map` method to transform the response into a json object. So when our ajax call is completed, a json object is placed in the observable and then pushed into our callback function

```
getSpotifyData(){  
  return this._http  
    .get("https://api.spotify.com/v1/search?q=jason&type=artist")  
    .map(res => res.json());  
}
```

When we run our app in the browser, we get the following result from the server (fig. 9.4.1).

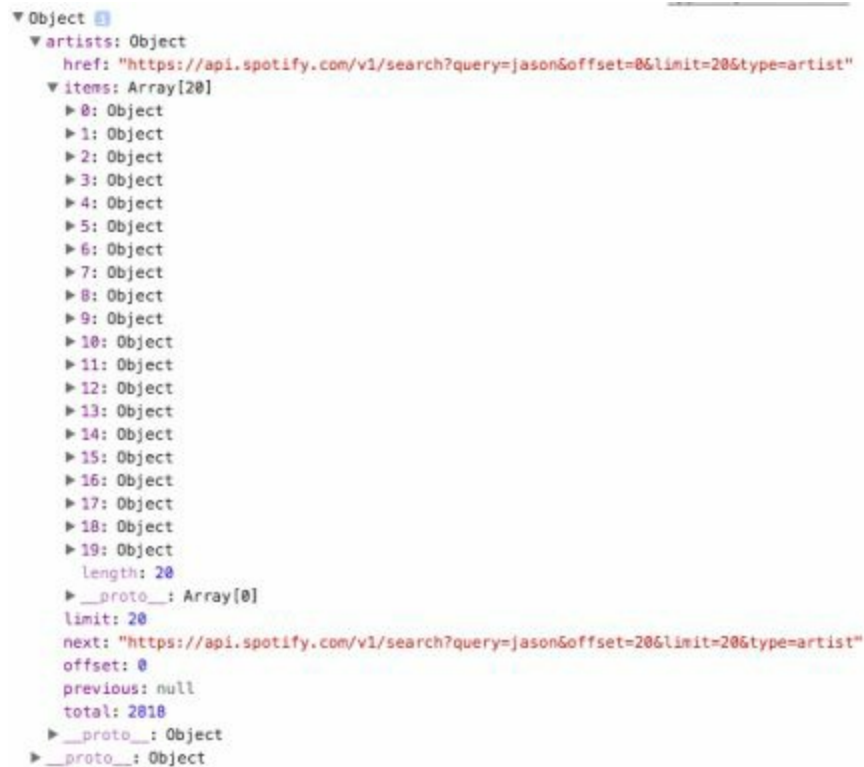


figure 9.3.1

Our json object is a single object containing an artists object containing an items array of size 20 each representing the data of an artist (fig. 9.3.2).



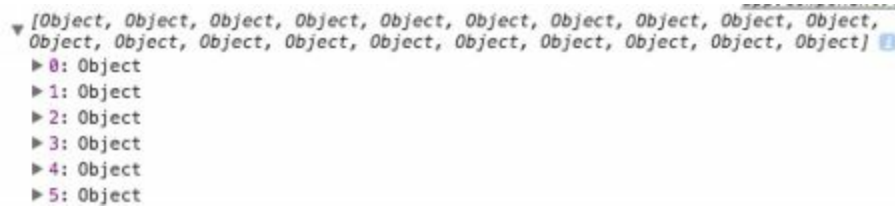
figure 9.3.2

Each artist object has properties followers , images , name , popularity and so on.

To get the items array direct, (since that is the data we want), we can further specify it like below

```
this._spotifyService.getSpotifyData()
    .subscribe(data => console.log(data.artists.items));
```

Doing so will get us the items array objects straight like in figure 9.3.3



```
[Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object, Object]
> 0: Object
> 1: Object
> 2: Object
> 3: Object
> 4: Object
> 5: Object
```

figure 9.3.3

9.4 Component's Lifecycle Hooks

Even though our code currently works, it doesn't follow best practises. We are currently calling the server in the constructor of the app component. As a best practise, constructors should be lightweight and should not contain any costly operations making it easier to test and debug. So where should we move our code to?

Components have a lifecycle which is managed by Angular. There are lifecycle hooks which we can tap into during key moments in the component's lifecycle. To do this, we need to implement one or more of the following interfaces in the component.

OnInit
OnDestroy
DoCheck
OnChanges
AfterContentInit
AfterContentChecked
AfterViewInit
AfterViewChecked

Each of the interfaces has a method that we can implement in our component. When the right moment arrives, Angular will call these methods. For example, we can implement the OnInit interface to be notified when a component is first instantiated.

By convention, the name of the method is the same name as the interface with the prefix ng . For e.g. ngOnInit .

We will implement the ngOnInit() method. This method will be called when Angular instantiates our component. In terms of lifecycle, it is called after the constructor. So in the constructor, we do lightweight and basic initialization and if we need to call the server, we do it in ngOnInit . So we shift the code to call Spotify from the constructor to ngOnInit as shown below.

```
export class AppComponent {
```

```
constructor(private _spotifyService: SpotifyService){  
  
}  
  
ngOnInit() {  
  this._spotifyService.getSpotifyData()  
    .subscribe(data => console.log(data.artists.items));  
}  
  
}
```

9.5 Showing a Loader Icon

When getting content from a server, it is often useful to show a loading icon to the user. To do so, in app component, create a variable called `isLoading` and set it to `true` like in the below code.

```
export class AppComponent {  
  isLoading = true;  
  ...  
}
```

Next, in the `subscribe` method, set `isLoading` to `false` because at this point, we get the results from the server and loading is finished.

```
ngOnInit() {  
  this._spotifyService.getSpotifyData()  
    .subscribe(data => {  
      this.isLoading = false;  
      console.log(this.artists);  
    });  
}
```

Lastly, in the template, add a `div` that shows the loading icon. We use `*ngIf` to make the `div` visible only when the component is loading.

```
template: `  
  <div *ngIf="isLoading">Getting data...</div>  
`,
```

If you load your app in the browser, you should see the “Getting data” message being displayed for a short moment before data from the server is loaded.

We will now replace the “Getting data” message with the loading icon. To get the loading icon, *google* ‘font awesome’ and the first result should be <http://fontawesome.io/>. Font awesome is a library similar to glyphs that gives us many useful icons. Go to ‘Get Started’ and follow the instructions to copy the embed code and paste it into the head section of `index.html`.

Back in app component, replace the message with the icon like in the below code.

```
template: `  
  <div *ngIf="isLoading">
```

```
<i class="fa fa-spinner fa-spin fa-3x"></i>
</div>
```

To add a *font awesome* icon, we use the `<i>` tag. `fa` is the base class for all font awesome icons. `fa-spinner` renders the spinner icon. `fa-spin` adds the animation to it. `fa-3x` makes the icon three times bigger so that it is easier to see.

9.6 Implementing a Spotify Results Display Page

We will now implement a page which displays our Spotify data nicely like in figure 9.6.1. To do so, we use the Bootstrap Media Object component from <http://getbootstrap.com/components/#media> as what we have done previously. We will be copying the markup from getbootstrap, pasting it into our component and filling the missing parts with interpolation strings.

Spotify Results



figure 9.6.1

Firstly, note that the images that we get from Spotify are of different height and width. To make all images of different height and width display the same in our page for e.g. 100px by 100px, we define the following inline css style class `img`.

```
@Component({
  selector: 'app-root',
  styles: [
```

```
.img {
  position: relative;
  float: left;
  width: 100px;
  height: 100px;
  background-position: 50% 50%;
  background-repeat: no-repeat;
  background-size: cover;
}
`],
```

In the template, we will be copy the markup from getbootstrap and paste it into our component as shown below.

```
template: `
<h3>Spotify Results</h3>
<div *ngIf="isLoading">
  <i class="fa fa-spinner fa-spin fa-3x"></i>
</div>
<div *ngFor="let artist of artists" class="media">
  <div class="media-left">
    <a href="#">
      
    </a>
  </div>
  <div class="media-body">
    <h4 class="media-heading">{{ artist.name }}</h4>
    Followers: {{ artist.followers.total }}
    Popularity: {{ artist.popularity }}
  </div>
</div>
`,
```


Code Explanation

```
<div *ngIf="isLoading">
  <i class="fa fa-spinner fa-spin fa-3x"></i>
</div>
```

We have added the loading icon as implemented previously.

```
<div *ngFor="let artist of artists" class="media">
```

We then apply `*ngFor` to repeat the media object for each artist we get from Spotify.

We then add four string interpolations inside the template. The artist's name, image, number of followers and her popularity. Note that Spotify displays larger image sizes in the first few elements of the image array. I therefore use the third element i.e. `artist.images[2]?.url` for a reasonably appropriate image size.

You can of course change this to use the first or second element if you want. Not all artist however have images. And you might get an error if you try to display an artist that does have an image array. To handle this, we add the `?` operator after the field to specify that it is optional.

```
<div *ngFor="let artist of artists" class="media">
  <div class="media-left">
    <a href="#">
      
    </a>
  </div>
  <div class="media-body">
    <h4 class="media-heading">{{ artist.name }}</h4>
    Followers: {{ artist.followers.total }}
    Popularity: {{ artist.popularity }}
  </div>
</div>
```

In our `*ngFor`, we state `let artist of artists`. But where do we get our artists array instantiated and assigned with content? We declare it in app component `artists = []` as shown below. We then subscribe to our Observable returned from our Spotify service and assign the returned result to artists array. Note that we assign it with `data.artists.items` as this is the artists item array structured in the json response.


```

export class AppComponent {
  isLoading = true;

  artists = [];

  constructor(private _spotifyService: SpotifyService){

  }

  ngOnInit() {
    this._spotifyService.getSpotifyData()
      .subscribe(data => {
        this.isLoading = false;
        this.artists = data.artists.items;
        console.log(this.artists);
      });
  }
}

```

If you run your app now, you should get a similar page as shown below.

Spotify Results



Jason Aldean

Followers: 962763 Popularity: 76



Jason Derulo

Followers: 2072640 Popularity: 82



Jason Mraz

Followers: 1418103 Popularity: 77



Jason Isbell

Followers: 76522 Popularity: 59



Jason Weaver

Followers: 3171 Popularity: 58

9.7 Adding an Input to Spotify Results Display Page

We are currently hard-coding our search term to `jason` in the url used to request from Spotify. We will now use the search input that we have implemented in chapter 8 and combine it with our code here so that as a user types in her search terms, the result can be displayed (figure 9.7.1).



figure 9.7.1

In chapter 8, we subscribe to an Observable stream from our input in the constructor of app component as shown below.

```
constructor(){
  this.searchControl.valueChanges
    .filter(text => text.length >= 3)
    .debounceTime(400)
    .distinctUntilChanged()
    .subscribe(value => {
      console.log(value);
    });
}
```

As mentioned earlier, such code should instead be placed in `ngOnInit` since the constructor should be lightweight. Also, we will move the code to subscribe to our Spotify Service into the callback function code block of `valueChanges` (see below).

```
ngOnInit() {
  this.searchControl.valueChanges
    .filter(text => text.length >= 3)
    .debounceTime(400)
    .distinctUntilChanged()
    .subscribe(value => {
      // insert call to spotify service here
    });
}
```

```
});
}
```

The final app component code will look like below. Note that we initialize `isLoading` to false at first since no call to Spotify is made at the beginning. Once we get a notification from `valueChanges` Observable, we then set `isLoading` to true just before the call to `getSpotifyData` to show the loading icon. Once we get notified of results from our spotify service Observable, we set `isLoading` to false to hide the loading icon.

```
export class AppComponent {
  searchControl = new FormControl();
  isLoading = false;
  artists = [];

  constructor(private _spotifyService: SpotifyService){
  }

  ngOnInit() {
    this.searchControl.valueChanges
      .filter(text => text.length >= 3)
      .debounceTime(400)
      .distinctUntilChanged()
      .subscribe(value => {
        this.isLoading = true;
        this._spotifyService.getSpotifyData(value)
          .subscribe(data => {
            this.isLoading = false;
            this.artists = data.artists.items;
          });
      });
  });
}
```

Finally, add the `<input>` tag to the template at the top:

```
template: `
  <input class="form-control" type="search"
    [formControl]="searchControl">
  <h3>Spotify Results</h3>
  <div *ngIf="isLoading">
    <i class="fa fa-spinner fa-spin fa-3x"></i>
  </div>
  <div *ngFor="let artist of artists" class="media">
    <div class="media-left">
      <a href="#">
        
```

```

        </a>
    </div>
    <div class="media-body">
        <h4 class="media-heading">{{ artist.name }}</h4>
        Followers: {{ artist.followers.total }}
        Popularity: {{ artist.popularity }}
    </div>
</div>
`
,

```

You can now see Spotify results displayed as you key in your search terms. And remember, because of the operators that we have earlier applied to our Observable, we do not send unnecessary multiple requests to Spotify.

Summary

In the chapter, we learned how implement a Spotify Search application by connecting our Angular apps to the server RESTful api using Observables, Http, component lifecycle hooks and display loader icons.

CHAPTER 10: BUILDING SINGLE PAGE APPS WITH ROUTING

We have so far covered components, directives and services. But what if we have multiple views that a user needs to navigate from one to the next? In this chapter, we will explore **Routers** that provide screen navigation in our Single Page Application.

We are familiar with navigating websites. We enter a URL in the address bar and the browser navigates to a corresponding page. We click links on the page and the browser navigates to a new page. We click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages we've seen.

The Angular Router borrows from this model. It interprets a browser URL as an instruction to navigate to a client-generated view and can also pass optional parameters along to the supporting view component to help it decide what specific content to present.

We can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. We can also navigate imperatively when the user clicks a button, selects from a drop box, or from other user generated events. And because the router logs activity in the browser's history journal, so the back and forward buttons work as well.

In this chapter, we will extend our project from chapter 9 to add routing to navigate between Home, Spotify, User Signup Form components.

10.1 Enabling Routing

The first step to building a Single Page application is to enable routing. We need to ensure that we have set the `<base href="/">` base url in our index.html . Angular will use this to compose relative urls. If you have created your Angular project using the Angular CLI, `<base href="/">` should have been added for you under the head element in index.html as show below:

```
<head>
  <meta charset="utf-8">
  <title>Angular2Firstapp</title>
  <base href="/">
```

The `/` means that our Angular app is currently in the application root. If your application has a lot of modules, your directory might look something like the below:

```
/users
  /app
  index.html
/posts
  /app
  index.html
/albums
  /app
  index.html
```

The base href for albums index.html will then be `<base href="/albums/">` .

10.2 Configuring Routes

After enabling routing, we need to define our routes. We define our routes in a separate new file. In your project from chapter 9, add a new file `app.routing.ts` with the below code:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { SpotifyComponent } from './spotify.component';

export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: '**', component: NotFoundComponent}
]);
```

Code Explanation

`app.routing.ts` contains our route definitions.

```
import { Routes, RouterModule } from '@angular/router';
```

We import `Routes` and `RouterModule` from Router library which provide the essential routing functionalities.

```
export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: '**', component: NotFoundComponent}
]);
```


RouterModule has a method forRoot which takes an array of Route definition objects. forRoot returns a module object and we assign it to variable routing . We need to export routing so that we can later import it in App Module. Note that routing is declared as a const which is a good practise so that no one will modify our routes making our application more reliable.

Wethen pass in our array of Route definition objects into the forRoot method. Each route definition associates a path to a component. Each route definition has at least two properties, path , which is the unique name we assign to our route, and component which specifies the associated component.

In our route definition, we have specified three components. HomeComponent , NotFoundComponent and our previously developed SpotifyComponent . We have not yet created HomeComponent and NotFoundComponent . So create the below components in app folder now.

home.component.ts

```
import { Component } from '@angular/core';
```

```
@Component({
  template: '<h1>Home</h1>'
})
export class HomeComponent {
}
```

notfound.component.ts

```
import { Component } from '@angular/core';
```

```
@Component({
  template: `
    <h1>Not Found</h1>
  `
})
export class NotFoundComponent { }
```

You will realize that HomeComponent and NotFoundComponent are very basic components that simply displays a message. This is for the purpose of illustrating navigating to different views.

Back in app.routing.ts , we import the components we will use in our route definitions as shown below in **bold**.

```
...
import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { SpotifyComponent } from './spotify.component';
...
```

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: '**', component: NotFoundComponent}
];
```

Now, our route definition tells Angular that:

- if the path changes to "", Angular should create an instance of HomeComponent and render it in the DOM.
- if the path changes to 'spotify', Angular should create an instance of SpotifyComponent and render it in the DOM.
- if a user navigates to a route that we have not defined, the path '**' is a wildcard that catches all invalid routes and directs to NotFoundComponent .

Next, in app.module.ts , we have to import and add our new components HomeComponent and NotFoundComponent to declarations array to declare that they are part of AppModule . We have to also import routing into the imports array as routing is a module which AppModule is dependent on.

The below lines in bold illustrate this.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';

import { AppComponent } from './app.component';
import { ReactiveFormsModule } from '@angular/forms';

import { UserFormComponent } from './user-form.component';
import { UserFormReactiveComponent } from './user-form-reactive.component';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { SpotifyComponent } from './spotify.component';
```

```
import { routing } from './app.routing';
```

```
@NgModule({  
  declarations: [  
    AppComponent,  
    HomeComponent,  
    NotFoundComponent,  
    SpotifyComponent  
  ],  
  imports: [  
    BrowserModule,  
    FormsModule,  
    ReactiveFormsModule,  
    HttpClientModule,  
    routing  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

10.3 Router Outlet and Links

Router Outlet

To specify where we want Angular to render our requested component when the user clicks on a link, we specify `<router-outlet></router-outlet>` in the DOM. For example in `app.component.ts` ,

```
import { Component } from '@angular/core';
```

```
@Component({  
  selector: 'app-root',  
  template: `  
    <h1>Hello</h1>  
    <router-outlet></router-outlet>  
  `,  
})  
export class AppComponent {  
}
```

Router Links

Having defined and configured our routes in `app.routing.ts` , we can now add our

navigation links to Home and Spotify component. In app.component.ts , add the below codes in bold.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <ul>
      <li><a routerLink="">Home</a></li>
      <li><a routerLink="spotify">Spotify</a></li>
    </ul>
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {
}
```

Code Explanation

```
<li><a routerLink="">Home</a></li>
<li><a routerLink="spotify">Spotify</a></li>
```

Note that we use routerLink directive instead of href to declare path to our routes. If we implement links using the traditional href way like below,

```
<li><a href="#">Home</a></li>
<li><a href="#">Spotify</a></li>
```

the href attribute will cause a full page reload which is contrary to the idea of Single Page Applications. In an SPA, we want our application to be loaded only once, and as we click on different links, only a part of the page is refreshed with the content of the target page. This results in much faster loading of pages.

We thus replace href with the routerLink directive and supply the name of the target route.

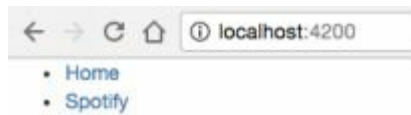
```
<li><a routerLink="">Home</a></li>
<li><a routerLink="spotify">Spotify</a></li>
```

routerLink tells our routing component to navigate the user to the target route specified. The routing component finds the route definition with that name. It will then create an instance of the component and render it in the router-outlet element.

And if we try a non-existent route, we get a 'not found' page because we have earlier declared the wildcard path to direct to NotFoundComponent .

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: '**', component: NotFoundComponent}
];
```

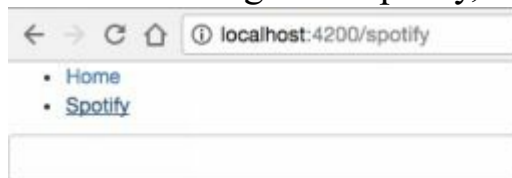
If we run our app now, we'll get a view like in figure 10.3.1.



Home

figure 10.3.1

And if we navigate to Spotify, we get the view like in figure 10.3.2



Spotify Results

figure 10.3.2

Improving the Look of our Navbar Component

Our navbar currently does not look very professional. We will use the navbar component from **getbootstrap** (<http://getbootstrap.com/components/#navbar>) to beautify our navigation bar. Use the default *navbar* markup in the template of navbar component. The default navbar renders a complex navbar with many complex items like a form. Get rid of all the unnecessary stuff (like dropdown and search) and keep only the Home and Spotify links.

Below shows the code for app.component.ts with the navbar component markup in template copied from getbootstrap. Unnecessary navbar elements like dropdown and search have been removed to result in a cleaner code. You can see the code for the router links in **bold**.

app.component.ts

```
import { Component } from '@angular/core';
```

```

@Component({
  selector: 'app-root',
  template: `
    <nav class="navbar navbar-default">
      <div class="container-fluid">
        <!-- Brand and toggle get grouped for better mobile display -->
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed"
            data-toggle="collapse" data-target="#bs-example-navbar-collapse-1"
            aria-expanded="false">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a class="navbar-brand" routerLink="">ngProject</a>
        </div>

        <!-- Collect the nav links, forms, and other content for toggling -->
        <div class="collapse navbar-collapse"
          id="bs-example-navbar-collapse-1">
          <ul class="nav navbar-nav">
            <li><a routerLink="" routerLinkActive="active">Home</a></li>
            <li><a routerLink="spotify"
              routerLinkActive="active">Spotify</a></li>
          </ul>
        </div><!-- /.navbar-collapse -->
      </div><!-- /.container-fluid -->
    </nav>
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {
}

```

When you run your app now, you should get a more professional navigation bar like in figure 10.3.3.

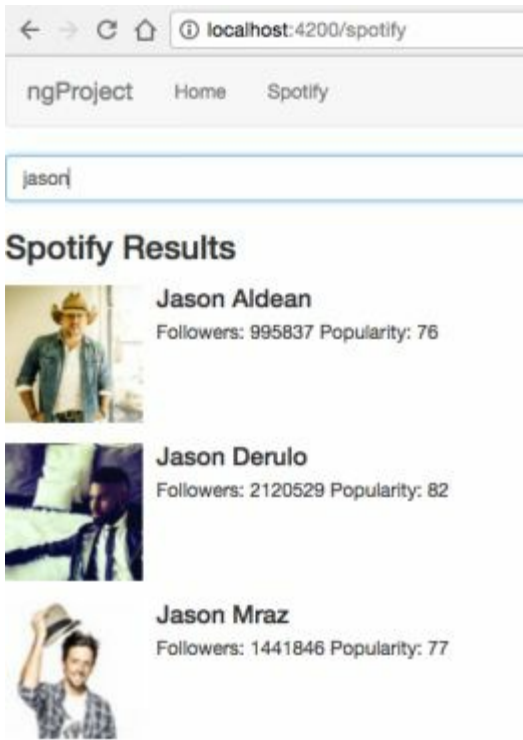


figure 10.3.3

And when you try to enter in an unspecified url, you get the `NotFoundComponent` rendered like in figure 10.3.4

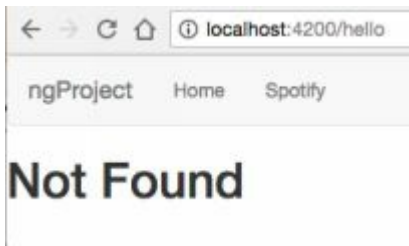


figure 10.3.4

Adding a new Router

Now let's try adding a new link called **Sign Up** that will take us to the User Form that we have implemented earlier in chapter six (figure 10.3.5).

User Form

First Name

Email

Country

Submit

figure 10.3.5

To do so, we add a new routerLink signup in App Component template.

```
<ul class="nav navbar-nav">
  <li><a routerLink="" routerLinkActive="active">Home</a></li>
  <li><a routerLink="spotify" routerLinkActive="active">Spotify</a></li>
  <li><a routerLink="signup" routerLinkActive="active">Sign Up</a></li>
</ul>
```

Then in App Module, make sure that you have imported UserFormComponent and add it to declarations array as shown below.

```
import { UserFormComponent } from './user-form.component';
...
```

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    NotFoundComponent,
    SpotifyComponent,
    UserFormComponent
  ],
```

Next in app.routing.ts, import UserFormComponent and add the path to the array in RouterModule.forRoot as shown below.

```
import { HomeComponent } from './home.component';
...
import { UserFormComponent } from './user-form.component';
```



```
export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: 'signup', component: UserFormComponent},
  {path: '**', component: NotFoundComponent}
]);
```

10.4 Route Parameters

We will now illustrate how to create routes that takes in route parameters. Why do we need this? For example, from the Spotify results page, we want to navigate to a page to see the details of a specific Spotify artist, we can pass in the information via route parameters.

In `app.routing.ts` , we add a route that takes in two route parameters as shown below in **bold**.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { SpotifyComponent } from './spotify.component';
import { UserFormComponent } from './user-form.component';

import { ArtistComponent } from './artist.component';

export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'spotify', component: SpotifyComponent},
  {path: 'spotify/artist/:id/:name', component: ArtistComponent},
  {path: 'signup', component: UserFormComponent},
  {path: '**', component: NotFoundComponent}
]);
```

Code Explanation

We first import the `ArtistComponent` which we will implement later. The `ArtistComponent` simply displays some information about a specific artist.

Next, we add a route

```
{path: 'spotify/artist/:id/:name', component: ArtistComponent}
```

`/:id/:name` represents the `id` route parameter and the `name` route parameter. If we want to pass in only one parameter for e.g. `id` , it will be just `spotify/artist/:id` . We can pass in

multiple parameters (more than two) if we want to.

With this route, whenever we navigate to a url for e.g.

<http://localhost:4200/spotify/artist/3Q8wgwyVVv0z4UEh1HB0KY/Jason%20Isbell>

Angular will render the ArtistComponent with the parameter id [3Q8wgwyVVv0z4UEh1HB0KY](http://localhost:4200/spotify/artist/3Q8wgwyVVv0z4UEh1HB0KY) and name [Jason%20Isbell](http://localhost:4200/spotify/artist/3Q8wgwyVVv0z4UEh1HB0KY/Jason%20Isbell).

You might ask, why is our route `spotify/artist/:id/:name` and not `artist/:id/:name`? That is because our Spotify search results is displayed in <http://localhost:4200/spotify/>. If our search results is displayed in the root i.e. <http://localhost:4200/>, then our route will be `artist/:id/:name`. We will explore different techniques to handle such routing later in the **Feature Routes** section.

Specifying Route Parameters

Next, in the template of `spotify.component.ts`, we add the line in **bold**.

```
template: `
  <input class="form-control" type="search"
    [formControl]="searchControl">
  <h3>Spotify Results</h3>
  <div *ngIf="isLoading">
    <i class="fa fa-spinner fa-spin fa-3x"></i>
  </div>
  <div *ngFor="let artist of artists" class="media">
    <div class="media-left">
      <a [routerLink]="['artist',artist.id, artist.name]">
        
      </a>
    </div>
    <div class="media-body">
      <h4 class="media-heading">{{ artist.name }}</h4>
      Followers: {{ artist.followers.total }}
      Popularity: {{ artist.popularity }}
    </div>
  </div>
`,
```

Code Explanation

```
<a [routerLink]="['artist',artist.id, artist.name]">
  
```


We add a `routerLink` to the artist image of each search result. When a user clicks on the artist image, she will be routed to `ArtistComponent` with parameters `id` and `name` .

```
<a [routerLink]="['artist',artist.id, artist.name]">
```

We use property binding syntax to bind our route parameters array to `routerLink` .

`['artist',artist.id, artist.name]` is our route parameters array. The first element is a string which contains our path `artist` . The second and third element are our route parameter values. If our route had more parameters, we can add them like `['artist',artist.id, artist.name, artist.popularity]`

Retrieving Route Parameters

Next, we create `ArtistComponent` that shows the details of a particular artist. In our case, we will just show the `id` and `name` of the artist. Create and fill in `artist.component.ts` with the below code.

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'artist',
  template: `
    <h1>Artist Id: {{ id }}</h1>
    <h2>Artist Name: {{ name }}</h2>
  `,
})
export class ArtistComponent implements OnInit {
  id;
  name;
  subscription;

  constructor(private _route: ActivatedRoute){
  }

  ngOnInit(){
    this.subscription = this._route.params.subscribe(params => {
      this.id = params["id"];
      this.name = params["name"];
    })
  }
}
```

Code Explanation

```
@Component({
  selector: 'artist',
  template: `
    <h1>Artist Id: {{ id }}</h1>
    <h2>Artist Name: {{ name }}</h2>`
})
```

In the template, we use string interpolation to display id and name . But how do we get the route parameters?

```
constructor(private _route: ActivatedRoute){
}
```

First, we use dependency injection to get an instance of ActivatedRoute . ActivatedRoute contains route information of a component and we subscribe to its params method to get our route parameters.

```
ngOnInit(){
  this.subscription = this._route.params.subscribe(params => {
    this.id = params["id"];
    this.name = params["name"];
  })
}
```

We implement the ngOnInit method (remember to implement OnInit in the class definition) and in it subscribe to _route.params which returns an Observable. We then get the value of the parameters using:

```
this.id = params["id"];
this.name = params["name"];
```

Additionally, to improve on memory, we can implement ngOnDestroy() so that we remove the subscription object from memory when this component instance is destroyed. See the complete code below with the OnDestroy code in **bold**.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
```

```
@Component({
  selector: 'artist',
  template: `
    <h1>Artist Id: {{ id }}</h1>
    <h2>Artist Name: {{ name }}</h2>
  `,
  ngOnDestroy() {
  },
})
```

```

export class ArtistComponent implements OnInit, OnDestroy {
  id;
  name;
  subscription;

  constructor(private _route: ActivatedRoute){
  }

  ngOnInit(){
    this.subscription = this._route.params.subscribe(params => {
      this.id = params["id"];
      this.name = params["name"];
    })
  }

  ngOnDestroy(){
    this.subscription.unsubscribe();
  }
}

```

Lastly, remember to import ArtistComponent and add it to declarations in app.module.ts as shown below in **bold**.

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
...

```

```

import { ArtistComponent } from './artist.component';

```

```

import { routing } from './app.routing';

```

```

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    NotFoundComponent,
    SpotifyComponent,
    UserFormComponent,
    ArtistComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    routing

```

```

],
providers: [],
bootstrap: [AppComponent]
}))
export class AppModule { }

```

10.5 Imperative Navigation

Suppose we want to redirect a user to another page upon clicking a button or upon clicking submit in a form. In such a case, we cannot use routerLink directive. Instead, we need to talk to the router object directly and this is what we called imperative or programmatic navigation.

To implement this, add the below codes in **home.component.ts**

```

import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  template: `
    <h1>Home</h1>
    <button (click)="onClick()">Sign Up</button>
  `
})
export class HomeComponent {

  constructor(private _router: Router){

  }

  onClick(){
    this._router.navigate(['signup']);
  }
}

```

Code Explanation

```

<button (click)="onClick()">Sign Up</button>

```

In the template, we have a **Sign Up** button and we do event binding to bind it to the onClick() method.

```

constructor(private _router: Router){
}

```

Next, we import and inject Router into the constructor of HomeComponent .

```
onClick(){
  this._router.navigate(['signup']);
}
```

In the `onClick` method, we call the `navigate` method of `Router` which takes in a route parameters array similar to the routes we have implemented earlier. The first element of the array will be the name of the target route and we supply any parameters in the second element of the array.

10.6 Route Guards

In a SPA, we may need to protect routes for example, preventing users from accessing areas that they're not allowed to access, or asking them for confirmation when leaving a certain area. Angular's router provides Route Guards that try to solve this problem.

There are two interfaces `CanActivate` and `CanDeactivate` which we can use to protect our routes. We use `CanActivate` to control if a user is allowed to navigate to a path, where certain pages are allowed only to logged in users or users with certain permissions or role. We use `CanDeactivate` to display a confirmation box to the user and ask if they really want to navigate away for example, if they have entered values into a form and tries to navigate away before saving the changes.

We refer to `CanActivate` and `CanDeactivate` as Route Guards because they act as guards to routes. We implement Route Guards in a separate service class and apply this service class on the routes we want to guard.

RouteGuard CanActivate

Suppose we want to let users access the User Form page only if they have logged in. In app, create a new file `auth-guard.service.ts` with the following codes.

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';

@Injectable()
export class AuthGuard implements CanActivate {

  canActivate(){
    return false;
  }

}
```

In the above code, we have currently set `canActivate()` to always return `false` and user will therefore never be able to access the page with this guard applied.

app.routing.ts

Next in `app.routing.ts` , add the codes in **bold**.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { UserFormComponent } from './users/user-form.component';

import {AuthGuard} from './auth-guard.service';

export const routing = RouterModule.forRoot([
  {path: '/', component: HomeComponent},
  {path: 'signup',
    component: UserFormComponent,
    canActivate: [AuthGuard]},
  {path: '**', component: NotFoundComponent}
]);
```

Code Explnanaion

```
canActivate: [AuthGuard]}
```

We import and add the parameter `canActivate` to the `signup` route. `canActivate` takes in an array of guards, which means we can apply multiple guards to a given route if needed. If the first one returns `false`, the execution stops there. Otherwise, control is passed to the next guard. For now, we pass in an array with a single guard element `AuthGuard` .

If we run the app now, and you try to navigate to **Sign Up**, nothing will happen and you will remain on the current page. This is because our route guard is at work here preventing access.

User Login

Now, how do we know if a user is currently logged in or not? Back in chapter 7, we implemented a separate login service class for authentication. The class had a login method. We further enhance the login class by adding a logout method and a boolean field `isLoggedIn` that tells us if a user is logged in or not.

The complete code for login.service.ts in app folder is shown below.

```
import {Injectable} from '@angular/core';
import {CanActivate} from '@angular/router';

@Injectable()
export class LoginService {
  isLoggedIn = false;

  login(username, password){
    if(username === "jason" && password === "123")
      this.isLoggedIn = true;
    else
      this.isLoggedIn = false;

    return this.isLoggedIn;
  }

  logout(){
    this.isLoggedIn = false;
    return this.isLoggedIn;
  }
}
```

As shown previously, the login service class is a simple plain Typescript class used behind our login form component. The values of username and password come in from the login form. In a real world application, login() should call a remote service with username/password and await a true or false return value. In this instance to simplify things, we will not be calling a remote service but instead hard code a specific username and password for successful login.

For the logout method, we don't call a remote service because the server is not aware if a user is logged in or not. This will be the client's responsibility which falls in the scope of Angular.

Do note that this is a basic implementation of an authentication service and in a real world application, you would probably have additional security measures like using encryption libraries to encrypt username/password and so on. We will next use this in our route guard.

auth-guard.service.ts

Back in auth-guard.service.ts , add the following codes in **bold**.

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
```

```

import { LoginService } from './login.service';

@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private _loginService: LoginService,
               private _router:Router){
  }

  canActivate(){
    if(this._loginService.isLoggedIn)
      return true;

    // imperative navigation
    this._router.navigate(['login'])

    return false;
  }
}

```

We import Router and LoginService and use dependency injection to get an instance of these two classes. In canActivate() , we then check if isLoggedIn in loginService is true. If yes, return true and allow the route to continue the navigation. If isLoggedIn is false , navigate to the login page and return false so that that the request page remains inaccessible.

app.routing.ts

Next in app.routing.ts , we import and apply the auth-guard route guard to our routes for Home, Spotify, Artist and UserForm as shown below in **bold**.

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { SpotifyComponent } from './spotify.component';
import { UserFormComponent } from './user-form.component';
import { ArtistComponent } from './artist.component';
import { LoginComponent } from './login.component';

import { AuthGuard } from './auth-guard.service';

export const routing = RouterModule.forRoot([
  {path: "", component: HomeComponent, canActivate: [AuthGuard]},
  {path: 'spotify', component: SpotifyComponent, canActivate: [AuthGuard]},
  {path: 'spotify/artist/:id/:name', component: ArtistComponent, canActivate: [AuthGuard]},

```

```

    {path: 'signup', component: UserFormComponent, canActivate: [AuthGuard]},
    {path: 'login', component: LoginComponent},
    {path: '**', component: NotFoundComponent}
  ]);

```

AppModule

Lastly, remember to register AuthGuard and LoginService in the providers array in AppModule if you have not already done so.

Running the App

If you run your app now and try to access either Home, Spotify, Artist or UserForm, you will be directed to the login form. And when you sign in with 'jason' and '123', you will be able to access the pages.

If for any reason you cannot get your app to run, contact me at support@i-ducate.com.

RouteGuard CanDeactivate

We will illustrate implementing our CanDeactivate route guard on our login page. That is, when a user has entered values into the login form and tries to navigate away, we will prompt a pop up message 'Are you sure' first before confirming user decision to navigate away.

We implement our CanDeactivate route guard in a separate service class. In app, add a new file `prevent-unsaved-changes-guard.service.ts`.

```

import { CanDeactivate } from '@angular/router';
import { LoginComponent } from '../login.component';

export class PreventUnsavedChangesGuard implements CanDeactivate<LoginComponent> {

  canDeactivate(component: LoginComponent) {
    if (component.form.dirty)
      return confirm("Are you sure?");

    return true;
  }
}

```

We implement the `canDeactivate` method and in it, we check if the form's `dirty` property which indicates if any of its form controls have been filled in. If yes, pop up the confirmation box. Else let the navigation away continue.

app.routing.ts

Next in `app.routing.ts` , apply the guard `PreventUnsavedChangesGuard` to the login route.

```
...
import { PreventUnsavedChangesGuard } from './prevent-unsaved-changes-guard.service';

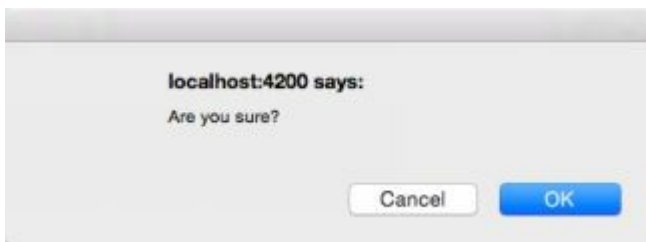
export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent, canActivate: [AuthGuard]},
  {path: 'signup', component: UserFormComponent, canActivate: [AuthGuard]},
  {path: 'login', component: LoginComponent, canActivate:
                                                                    [PreventUnsavedChangesGuard]},
  {path: '**', component: NotFoundComponent}
]);
```

AppModule

Remember to register `PreventUnsavedChangesGuard` in the providers array in `AppModule` .

Running the App

Now run the app and enter some values into the login form fields. Try navigating away and you should see the below pop up box confirmation your decision.



Summary

In this chapter, we see how to build single page apps with routing. We learnt how to define and configure routes, rendering requested component using the router outlet, providing router links, how to create routes with parameters, how to retrieve the parameters, using route guards for authorization with the `CanActivate` interface and preventing unsaved changes with the `CanDeactivate` interface.

We have covered a lot in this chapter. Contact me at support@i-ducate.com if you have not already to have the full source code for this chapter or if you encounter any errors with your code.

CHAPTER 11: STRUCTURING LARGE APPS WITH MODULES

The application we have built in chapter nine and ten so far has three main function areas. The Home page, the Search Spotify page and the Sign Up page. But once our app grows, maintenance becomes more important and challenging. It is then better to divide a large app into smaller parts each focusing on one specific functionality. Each part will have a group of highly related classes that work together to fulfil a specific function.

Gmail for example is a huge application. In it, we find different functionalities like Inbox, Contacts and Compose. Each function would have been formed from their own set of highly related classes which we term as one module.

In Angular, each application has at least one module, which is our main or root module **App Module**. We so far have been adding our components to this single root module.

But as our app grows, we should refactor this into smaller and more focused modules for better maintainability.

For example, in our application, our module directory will look something like:

```
/AppModule  
  /SpotifyModule  
  /UserModule  
  ...
```

11.1 NgModule Decorator

A module is really just a class decorated with the NgModule decorator. We already seen it in app.module.ts in chapter 10 and earlier chapters (shortened code snippet shown below).

```
import ...
```

```
@NgModule({  
  declarations: [  
    AppComponent,  
    HomeComponent,  
    NotFoundComponent,  
    SpotifyComponent,  
    UserFormComponent,  
    ArtistComponent  
  ],  
  imports: [  

```

```

    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    routing
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

As mentioned earlier, the imports array specify what other modules this present module depends on. We can see that the above AppModule depends on BrowserModule, FormsModule, ReactiveFormsModule, HttpClientModule and our routing Module.

The declarations array specify what components, directives and pipes are part of this module. We currently have AppComponent, HomeComponent, NotFoundComponent, SpotifyComponent, UserFormComponent, ArtistComponent as part of AppModule.

We specify the boot or entry module in the bootstrap array of our app. We have specified AppComponent as the entry point for our application. Note that this is only required in the root module AppModule and not needed in the sub-modules we create.

The providers array specify mainly service classes that we use through dependency injection.

So what is the benefit of listing components and services in these arrays? The benefit is that we don't have to individually import components for every other component in the same module. For e.g. in SpotifyComponent, we already have access to ArtistComponent without importing it since it is declared to be in the same module. So as long as components are in the same module, they will be available to one another in the module. This results in much cleaner code where we don't have to keep repeating similar import statements in multiple classes.

11.2 Refactoring

Our AppModule now is beginning to get quite huge and messy. Currently, all files are in one single module. So how do we re-structure it into smaller, focused modules like in the below structure?

```

/AppModule
/SpotifyModule
/UserModule
...

```

We will try to refactor some classes out from AppModule to

form SpotifyModule and UserModule . The related classes for SpotifyModule will be SpotifyComponent , ArtistComponent , SpotifyService .

The related classes for UserModule will be UserFormComponent and User .

HomeComponent and NotFoundComponent will remain in AppModule since they are generic to the application.

We create two new folders spotify and users in / app . So that the folder structure looks like below:

```
/app
  /spotify
  /users
...
```

Refactoring SpotifyModule

We first refactor for SpotifyModule . Move the below files from /app to /app/spotify

```
artist.component.ts
spotify.component.ts
spotify.service.ts
```

In /app/spotify , create spotify.module.ts with the below code.

```
import { NgModule }      from '@angular/core';
import { CommonModule }   from '@angular/common';
import { ReactiveFormsModule } from '@angular/forms';
import { RouterModule }   from '@angular/router';
import { HttpClientModule } from '@angular/http';

import { SpotifyComponent } from './spotify.component';
import { ArtistComponent }  from './artist.component';
import { SpotifyService }   from './spotify.service';

@NgModule({
  imports: [
    CommonModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule
  ],
  declarations: [
    SpotifyComponent,
    ArtistComponent
```



```

    ],
    exports: [
    ],
    providers: [
        SpotifyService
    ]
  })
  export class SpotifyModule {
  }

```

Code Explanation

Notice that `spotify.module.ts` is very much similar to `app.module.ts` except that it contains files specific to the Spotify Module which we store in a separate Spotify folder.

A difference is that we import `CommonModule` instead of `BrowserModule`. In an Angular app, only the root application module `AppModule` should import `BrowserModule`. `BrowserModule` provides services that are essential to launch and run a browser app. Feature modules (or sub-modules) should import `CommonModule` instead. They need the common directives and don't need to re-install the app-wide providers.

Refactoring UserModule

Next, we refactor for `UserModule`. Move the below files from `/app` to `/app/users`.

```

user.ts
user-form.component.ts
user-form.component.html

```

In `/app/users/`, create `user.module.ts` with the below code.

```

import { NgModule }      from '@angular/core';
import { CommonModule }   from '@angular/common';
import { FormsModule }   from '@angular/forms';

import { UserFormComponent } from './user-form.component';

@NgModule({
  imports: [
    CommonModule,
    FormsModule
  ],
  declarations: [
    UserFormComponent

```

```

    ],
    exports: [
    ],
    providers: [
    ]
  })
  export class UserModule {
  }

```

Refactoring AppModule

Because we have already imported `ReactiveFormsModule`, `HttpModule`, `RouterModule` in `SpotifyModule` and `FormsModule` in `UserModule`, we can remove them from `AppModule`.

Similarly, because we have imported `SpotifyComponent`, `ArtistComponent`, `SpotifyService` in `SpotifyModule` and `UserFormComponent`, `User` in `UserModule`, we can remove the import statements for these classes from `AppModule` as shown below. Instead, we import `SpotifyModule` and `UserModule`.

app.module.ts

```

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

```

```

import { AppComponent } from './app.component';
import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';

```

```

import { routing } from './app.routing';

```

```

import { SpotifyModule } from './spotify/spotify.module';
import { UserModule } from './users/user.module';

```

```

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    NotFoundComponent
  ],
  imports: [
    BrowserModule,
    SpotifyModule,
    UserModule,
    routing
  ],
  providers: [],
  bootstrap: [AppComponent]
})

```

```
  })  
  export class AppModule { }
```

Realize that after refactoring, AppModule becomes much smaller? We only make code changes to AppModule when we add new modules to it. And both SpotifyModule and UserModule can grow on its own. We can add new classes, components, pipes, directives to the them without impacting AppModule. This is the benefit of modularity.

Finally in app.routing.ts , do the below minor changes in **bold**.

```
import { NgModule } from '@angular/core';  
import { Routes, RouterModule } from '@angular/router';  
  
import { HomeComponent } from './home.component';  
import { NotFoundComponent } from './notfound.component';  
import { SpotifyComponent } from './spotify/spotify.component';  
import { UserFormComponent } from './users/user-form.component';  
  
import { ArtistComponent } from './spotify/artist.component';  
  
export const routing = RouterModule.forRoot([  
  {path: '', component: HomeComponent},  
  {path: 'spotify', component: SpotifyComponent},  
  {path: 'spotify/artist/:id/:name', component: ArtistComponent},  
  {path: 'signup', component: UserFormComponent},  
  {path: '**', component: NotFoundComponent}  
]);
```

11.3 Refactoring Routes

Although app.module.ts has been structured to be more modular, app.routing.ts still contains all the routes in a single file. If the number of routes increases to hundreds or thousands, app.routing.ts will become messy and unmaintainable. So just as we have refactored app.module.ts , we can also divide our routes into smaller more manageable routing files.

With this structure, instead of having a gigantic app.routing.ts with hundreds of routes, we will have a feature route file per module and app.routing.ts will delegate all the routing for that module to that feature routing file.

That is, we move routes in a feature area to its corresponding module. For example, we move the below two routes to their own Spotify Module routing file.

```
{path: 'spotify', component: SpotifyComponent},
```

```
{path: 'spotify/artist/:id/:name', component: ArtistComponent},
```

In Spotify folder, add a new file `spotify.routing.ts` with the below code. This will be the routing file for Spotify Module.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { SpotifyComponent } from './spotify.component';
import { ArtistComponent } from './artist.component';

export const spotifyRouting = RouterModule.forChild([
  {path: 'spotify', component: SpotifyComponent},
  {path: 'spotify/artist/:id/:name', component: ArtistComponent}
]);
```

app.routing.ts

`app.routing.ts` will have the below code. Note that the Spotify related routes have been removed resulting in much cleaner and modular code.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';
import { UserFormComponent } from './users/user-form.component';

export const routing = RouterModule.forRoot([
  {path: '', component: HomeComponent},
  {path: 'signup', component: UserFormComponent},
  {path: '**', component: NotFoundComponent}
]);
```

app.module.ts

`app.module.ts` will have the below code changes in **bold**.

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

import { AppComponent } from './app.component';
import { HomeComponent } from './home.component';
import { NotFoundComponent } from './notfound.component';

import { routing } from './app.routing';
import { spotifyRouting } from './spotify/spotify.routing'
```

```

import { SpotifyModule } from './spotify/spotify.module';
import { UserModule } from './users/user.module';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    NotFoundComponent
  ],
  imports: [
    BrowserModule,
    SpotifyModule,
    UserModule,
    spotifyRouting,
    routing
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Here, we import both our routing modules into AppModule .

Summary

In this chapter, we covered how to refactor our application as it grows, into smaller and more focused modules for better maintainability. We learnt how to declare a module with the NgModule decorator, how to refactor files into separate module folder structures and also how to refactor routes to their own modules.

CHAPTER 12: C.R.U.D. WITH FIREBASE

In this chapter, we will cover how to implement full C.R.U.D. operations in Angular with a backend server. A typical web application architecture consists of the server side and client side. This book teaches you how to implement the client side using Angular. The client side talks to a backend server to get or save data via RESTful http services built using server side frameworks like ASP.NET, Node.js and Ruby on Rails. We have explored this when we got data from the Spotify server in chapter nine.

Building the server side however is often time consuming and not in the scope of this course. In this chapter however, we will explore using Firebase as our backend server. Firebase is a very powerful backend platform for building fast and scalable real-time apps. It has been gaining popularity since 2015 and we will introduce it here, so don't worry if you have never worked with Firebase before.

With Firebase, we don't have to write server side code or design relational databases. Firebase provides us with a real-time, fast and scalable NoSQL database in the cloud and also a library to talk to this database. This allow us to focus on building our application according to requirements rather than debugging server side code.

You might ask, what is a NoSQL database? In contrast to relational databases which consists of tables and relationships, in a NoSQL database, we have a tree of JSON objects and each node in the tree can have a different structure. Because we do not have to maintain table schemas, NoSQL databases provide us with one thing less to worry about thereby increasing productivity. However, if you application involves lots of data aggregating, complex querying and reporting, a relational database might still be a better choice.

The aim of this chapter is to however illustrate create, read, update and delete functionality with Angular and Firebase integrated so that you can go on and create a fully working app. And if you choose to have a different backend server like ASP.NET, Node.js, the same principles will apply.

More on Firebase

Firebase is a real time database. which means that as data is modified, all connected clients are automatically refreshed in an optimised way. If one user adds a new item either through a browser or a mobile app, another user (again either through a browser or mobile app) sees the addition in real time without refreshing the page. Firebase of course provides more than just a real time database. It provides other services like Authentication, cloud messaging, disk space, hosting an analytics. You not only can

develop Angular apps with Firebase as backend, but also iOS, Android and web applications.

12.1 Using Firebase

We can use Firebase features for free and only pay when our application grows bigger. You can choose between a subscription based or ‘pay as you use’ model. Find out more at firebase.google.com/pricing.

Before adding Firebase to our Angular project, we need to first create a Firebase account. Go to firebase.google.com and sign in with your Google account.

Click **‘Get Started for Free’** to go to the Firebase console. In the console, click on **‘Create New Project’** (figure 12.1)

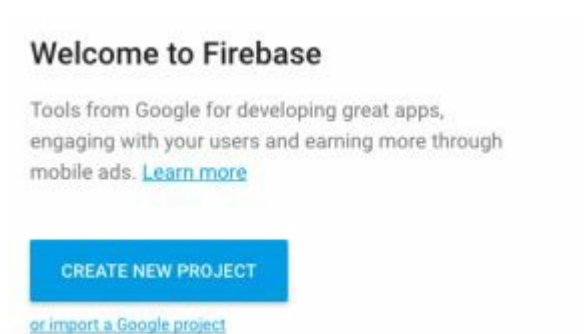


figure 12.1.1

Fill in the project name, country and click ‘Create Project’.

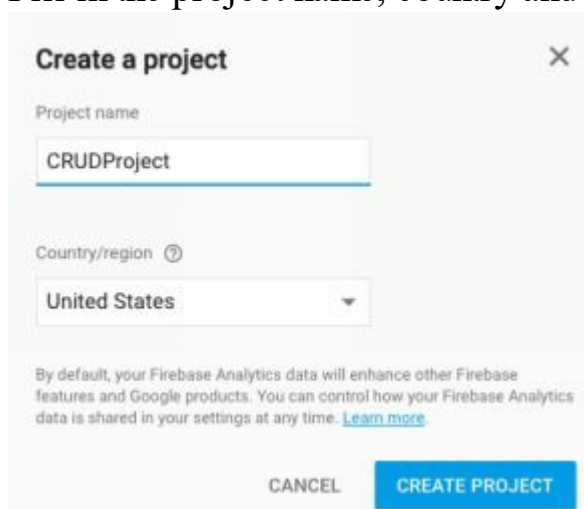


figure 12.1.2

In the Welcome screen, click on ‘Add Firebase to your web app’ (figure 12.3).



figure 12.1.3

You will see some configuration code that you need to add in your project (fig. 12.1.4).

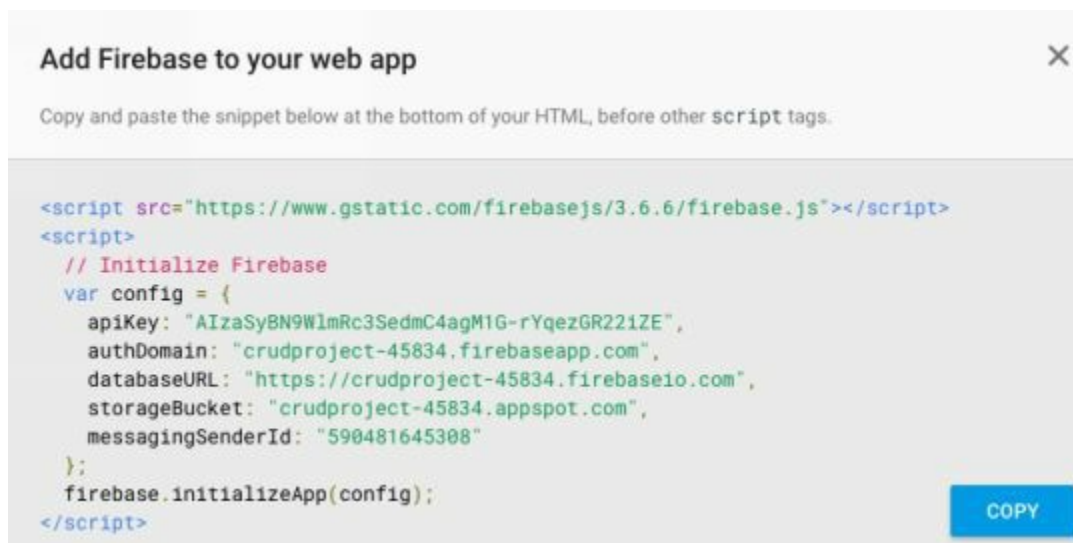


figure 12.1.4

Code Explanation

```
<script src="https://www.gstatic.com/firebasejs/3.6.4/firebase.js"></script>
```

This is a script reference to Firebase SDK. `firebase.js` gives us a library to work with firebase.

```
<script>
// Initialize Firebase
var config = {
  apiKey: "AIzaSyBN9WlmRc3SedmC4agM1G-rYqezGR22iZE",
  authDomain: "crudproject-45834.firebaseio.com",
  databaseURL: "https://crudproject-45834.firebaseio.com",
  storageBucket: "crudproject-45834.appspot.com",
  messagingSenderId: "590481645308"
};
firebase.initializeApp(config);
</script>
```


We have a config or configuration object with properties `apiKey`, `authDomain` (a subdomain under `firebaseapp.com`), `databaseUrl`, `storageBucket` (for storing files like photos, videos etc.) and `messagingSenderId` (used for sending push notifications).

12.2 Adding Firebase to our Angular App

To illustrate connecting Firebase to our Angular app, we will create a new project using Angular CLI (I have named my project `CRUDProject`).

```
ng new CRUDProject
```

We will next use `npm` to add `firebase` and another library called `angularfire2` to our project.

```
npm install firebase angularfire2 --save
```

After the installation, in `package.json`, an entry for `firebase` will have been added (see lines below in **bold**).

```
"dependencies": {  
  "@angular/common": "2.0.0",  
  "@angular/compiler": "2.0.0",  
  "@angular/core": "2.0.0",  
  "@angular/forms": "2.0.0",  
  "@angular/http": "2.0.0",  
  "@angular/platform-browser": "2.0.0",  
  "@angular/platform-browser-dynamic": "2.0.0",  
  "@angular/router": "3.0.0",  
  "angularfire2": "^2.0.0-beta.6",  
  "core-js": "^2.4.1",  
  "firebase": "^3.6.4",  
  "rxjs": "5.0.0-beta.12",  
  "ts-helpers": "^1.1.1",  
  "zone.js": "^0.6.23"  
},
```

(Note: At time of writing, the version for `"angularfire2": "^2.0.0-beta.6"`, and `"firebase": "^3.6.4"`)

app.module.ts

Next, we need to import `AngularFireModule` into our App Module. In `app.module.ts`, add the lines in **bold**. Note that the credential properties in `firebaseConfig` should be your own (copied from firebase console)

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';
import { AngularFireModule } from 'angularfire2';

import { AppComponent } from './app.component';

export const firebaseConfig = {
  apiKey: "AIzaSyBN9WlmRc3SedmC4agM1G-rYqezGR22iZE",
  authDomain: "crudproject-45834.firebaseio.com",
  databaseURL: "https://crudproject-45834.firebaseio.com",
  storageBucket: "crudproject-45834.appspot.com",
  messagingSenderId: "590481645308"
};

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    AngularFireModule.initializeApp(firebaseConfig)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {

}

```

Code Explanation

```
import { AngularFireModule } from 'angularfire2';
```

angularfire2 is a library that sits on top of firebase and makes it easier to build Angular 2 apps that use firebase as the backend as we will see shortly.

app.component.ts

Now to make sure that we have added firebase correctly to our project, go to app.component.ts and add the lines **bold**.

```

import { Component, OnInit } from '@angular/core';
import { AngularFire} from 'angularfire2';

```

```

@Component({
  selector: 'app-root',
  template: `
    <div>
    </div>
  `
})
export class AppComponent {
  constructor(af: AngularFire){
    console.log(af);
  }
}

```

Now, make sure that the lite web server is running, (by executing `ng serve`) and in the console, you should see the AngularFire object printed as shown below to prove that we have added Angular Fire correctly.

```

app.component.ts:12
AngularFire {firebaseConfig: Object, auth: AngularFireAuth,
database: AngularFireDatabase}

```

12.3 Working with a Firebase Database

Now let's look at our Firebase database. Go to console.firebase.google.com . Click on your project, and from the menu bar on the left, click on **Database**.

We will store our data here. If you have not worked with NoSQL databases before, you might find it odd in the beginning because there is no concept of tables or relationships here. Our database is basically a tree of key value pair objects. We basically store json objects here that map natively to json objects in javascript. So when working with a NoSQL database on the backend, we get a json object from the server and we simply display in on the client. Or we construct a json object on the client and send it to server and we store it as it is. There is no additional mapping needed i.e. from relational format to json format or vice-versa.

Click + to add a new child node to the tree. Each node has a name and a value. Value can be a primitive type like string, boolean, number or it can be a complex object.



The screenshot shows the Firebase Database console interface. At the top, there's a breadcrumb 'crudproject-45834' followed by a 'null' value and a close button. Below this, a form is displayed with two input fields: 'Name' containing 'name' and 'Value' containing 'Ervin Lim'. To the right of the 'Value' field is another close button. At the bottom of the form are two buttons: 'CANCEL' and 'ADD'.

When you click **Add**, a new node will be added.

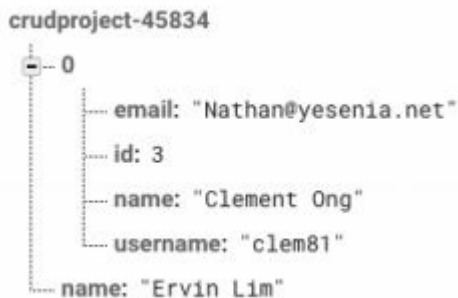
```

crudproject-45834
└── name: "Ervin Lim"

```

(Note that when you add a new child node, the child node gets highlighted in green and the parent node in yellow for a few seconds. If you try deleting a node, that node gets highlighted in red.)

Our values can also be complex objects. You can add complex objects by clicking on the + sign in Value of an existing child node. The below tree has a childnode 0 that contains further properties.



You can of course have complex objects in complex, for e.g.



Essentially, we have a hierarchy of key value pairs in a NoSQL database. We don't have tables and relationships. The modelling of objects and their relationships vital to an enterprise level application is beyond the scope of this book (but you can contact me at support@i-ducate.com to enquire about my upcoming book on in-depth Firebase).

In the next sections, we will illustrate with a simple example of user objects in our

NoSQL database.

12.4 Displaying List of Users

We will illustrate how to display a list of users. But before that, we need to have existing user data in Firebase. We will use users data from jsonplaceholder at <http://jsonplaceholder.typicode.com/>. jsonplaceholder provides with a fake online REST api and data for testing. So head to

<http://jsonplaceholder.typicode.com/users>

and save the json file. I have saved it as users.json . We can import this json file into our Firebase database by going to Database, click on the most right icon, and select 'Import JSON' (fig. 12.4.1).



figure 12.4.1

Browse to the user json file you saved and click 'Import' (fig. 12.4.2).

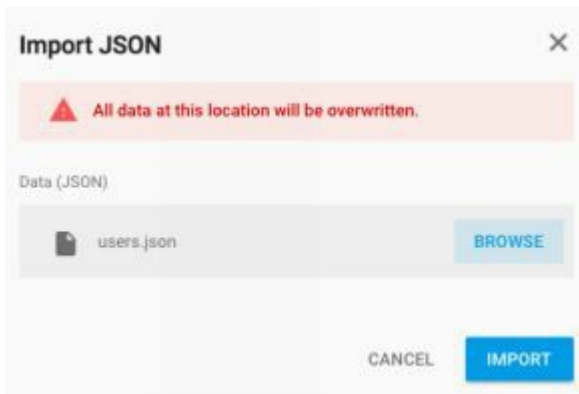


figure 12.4.2

The users data will be imported into firebase (fig. 12.4.3).



figure 12.4.3

user.component.ts

Next, we will create a user component to display our list of users. Create `user.component.ts` in `app` with the following code.

```
import { Component, OnInit } from '@angular/core';
import { AngularFire } from 'angularfire2';
```

```
@Component({
  selector: 'users',
  templateUrl: './user.component.html'
})
export class UserComponent {
  users;

  constructor(private af:AngularFire){
  }

  ngOnInit(){
    this.users = this.af.database.list('/');
  }
}
```

Code Explanation

```
import { AngularFire } from 'angularfire2';
```

We import AngularFire and get an instance of it from injecting it in our constructor.

```
ngOnInit(){
  this.users = this.af.database.list('/');
}
```

In `ngOnInit`, we specify the location of a node in firebase as an argument to the `list` method to retrieve our list of users. In our case, our list of users is at the root node and thus, we specify `'/'`. But say if our list of users is a child node under the parent node `'spotify'`, we would then have something like

```
this.users = this.af.database.list('/spotify');
```

You might be asking, isn't the code to list users too simple? How is that we can use the `Observable` directly here and not have to subscribe to an `rxjs Observable` like what we have done when we got data from Spotify? The answer is because `af.database.list` returns a `FirebaseListObservable` type which is firebase's own `Observable` that wraps around the standard `rxjs Observable`. `FirebaseListObservable` is an `rxjs Observable` internally but it wraps around it and provides additional methods which makes it easy for us to execute `create`, `read`, `update` and `delete` functions on it.

If we did not use the `FirebaseListObservable` directly, the old way would be to subscribe to it as what we have done earlier (in chapter nine) like the below:

```
ngOnInit(){
  this.subscription = this.af.database.list('/').subscribe(x =>{
    this.users = x;
  });
}

ngOnDestroy(){
  this.subscription.unsubscribe();
  // so that we do not consume memory ever increasingly
}
}
```

See how much simpler it is to use `FirebaseListObservable`?

user.component.html

Next, create `user.component.html` with the below code:

```
<h1>Users</h1>
<table class="table table-bordered">
  <thead>
    <tr>
```

```

<th>Username</th>
<th>Email</th>
<th>Edit</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr *ngFor="let user of users | async">
<td>{{ user.name }}</td>
<td>{{ user.email }}</td>
<td>
<a>
<i class="glyphicon glyphicon-edit"></i>
</a>
</td>
<td>
<a>
<i class="glyphicon glyphicon-remove"></i>
</a>
</td>
</tr>
</tbody>
</table>

```

Code Explanation

```
<table class="table table-bordered">
```

We use the bootstrap classes `table` and `table-bordered` to create a nice looking table for listing our users.

```

<a>
<i class="glyphicon glyphicon-edit"></i>
</a>
</td>
<td>
<a>
<i class="glyphicon glyphicon-remove"></i>
</a>

```

We also use `glyphicons edit` and `remove` for the edit and delete operations we will implement later.

```

<tr *ngFor="let user of users | async">
<td>{{ user.name }}</td>

```



```
<td>{{ user.email }}</td>
```

Notice that we have applied the async pipe in our `*ngFor` to display users. Because data in users arrive asynchronously, the async pipe subscribes to user (which is a `FirebaseObjectObservable`) and returns the latest value emitted. The async pipe marks the component to be checked for changes. It also removes the subscription once the component is destroyed (thus no longer needing `ngOnDestroy`).

app.routing.ts

Create the routing file `app.routing.ts` in `app` as shown below.

```
import { RouterModule } from '@angular/router';
import { UserComponent } from './user.component';

export const routing = RouterModule.forRoot([
  { path:'', component:UserComponent },
]);
```

It currently contain only one route which points to User component. We will extend the routes later to include the route to the User Add and Edit form.

app.module.ts

Add the below lines in **bold** to `app.module.ts` to import and declare that `UserComponent` belongs to App Module. We also import routing .

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/http';
import { AngularFireModule } from 'angularfire2';

import { AppComponent } from './app.component';
import { UserComponent } from './user.component';

import { routing } from './app.routing';

export const firebaseConfig = {
  apiKey: "AIzaSyC94rD8wXG0aRLTcG29qVGw8CFfvCK7XVQ",
  authDomain: "myfirstfirebaseproject-6da6c.firebaseio.com",
  databaseURL: "https://myfirstfirebaseproject-6da6c.firebaseio.com",
  storageBucket: "myfirstfirebaseproject-6da6c.appspot.com",
  messagingSenderId: "138019512918"
};

@NgModule({
```

```

declarations: [
  AppComponent,
  UserComponent,
],
imports: [
  BrowserModule,
  HttpClientModule,
  AngularFireModule.initializeApp(firebaseConfig),
  routing
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule {

}

```

app.component.ts

Back in app.component.ts , we add router-outlet into the template as shown below to render our content.

```

import { Component, OnInit } from '@angular/core';
import { AngularFire } from 'angularfire2';

@Component({
  selector: 'app-root',
  template: `
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {
  constructor(af: AngularFire){
    console.log(af);
  }
}

```

To make the code cleaner, we also remove the previous logging code (to see if we have installed AngularFire correctly).

index.html

Remember to add the bootstrap link in index.html as shown below to make sure that our table borders are rendered correctly.

```

<!doctype html>
<html>

```

```

<head>
  <meta charset="utf-8">
  <title>CRUDproject</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-
BVYiSiFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>

```

Setting Permissions for Read

If we run our app now, we will get an error saying something like “Permission denied. Client doesn’t have permission to access the desired data.” This is because in firebase, our firebase permission rules are currently configured as:

```

{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}

```

The above code is a json object that determines the rules for reading and writing of data. You can access these rules in firebase console, under **Database, Rules** tab. Essentially, the code is saying that read and write permission is only granted to those who are logged in or authenticated (auth != null). Because firebase authentication and authorisation is beyond the scope of this book, and to quickly get a fully working Angular app, we will set both read and write permissions to be public, where anyone can read or write to our database. (I will teach about firebase permissioning in a future book, contact me at support@i-ducate.com for more information)

So in firebase console in the Rules tab, edit the permission rules as shown below and click **Publish**. Note that whenever we make changes to our permission rules, we need to publish the changes.

```

{
  "rules": {
    ".read": true,
    ".write": true
  }
}

```

```
}  
}
```

Running your App

Now if you run your app, you should see a list of users rendered like in figure 12.4.4.

Users



















Username	Email	Edit	Delete
Ervin Howell	Shanna@melissa.tv		
Clementine Bauch	Nathan@yesenia.net		
Patricia Lebsack2	Julianne.OConner@kory.org2		
Chelsey Dietrich	Lucio_Hettinger@annie.ca		
Mrs. Dennis Schulist	Karley_Dach@jasper.info		
Kurtis Weissnat	Telly.Hoeger@billy.biz		
Nicholas Runolfsdottir V	Sherwood@rosamond.me		
Glenna Reichert	Chaim_McDermott@dana.io		
Jason Try	Jason Again		

figure 12.4.4

Now, try going back to the firebase console and add a new user node. When you go back to your Angular app, you will realize that the user list is refreshed automatically with the new node! Or if you delete a node from the firebase console, the list is refreshed to reflect the deletion as well. And that's the beauty of firebase. We achieved auto-refresh upon adding, updated, delete with just the single line of code below:

```
this.users = this.af.database.list('/');
```

If we were to try to do this without firebase, it would take a lot more code.

12.5 Adding a User

user.component.html

Next, we will implement adding a user to our app. First, add a button called **Add User** just before the user list in `user.component.html`. Decorate it with css button classes `btn` and `btn-primary` as shown below.

```
<h1>Users</h1>  
<button class="btn btn-primary" (click)="add()">Add</button>  
<table class="table table-bordered">  
...  

```

user.component.ts

When we click this button, we route to a new page with a form to add a new user. To create this route, implement the add() method in user.component.ts as shown below.

```
import { Component, OnInit } from '@angular/core';
import { AngularFire } from 'angularfire2';
import { Router } from '@angular/router';

@Component({
  selector: 'users',
  templateUrl: './user.component.html'
})
export class UserComponent {
  users;

  constructor(private af:AngularFire, private _router: Router){
  }

  ngOnInit(){
    this.users = this.af.database.list('/');
  }

  add(){
    this._router.navigate(['add']);
  }
}
```

app.routing.ts

In app.routing.ts , import and add the path to UserForm component as shown below. We will create UserForm component in the next section.

```
import { RouterModule } from '@angular/router';
import { UserComponent } from './user.component';
import { UserFormComponent } from './user-form.component';

export const routing = RouterModule.forRoot([
  { path:'', component:UserComponent },
  { path:'add',component:UserFormComponent }
]);
```

user.ts

We represent the model data behind our User form with the class User in user.ts . So add this class in app .

```
export class User{
  id: string;
  username: string;
  email: string;
}
```

user-form.component.ts

Next, create a new component `user-form.component.ts` that implements a model driven form with fields, username and email as shown below.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { AngularFire } from 'angularfire2';
```

```
import { User } from './user';
```

```
@Component({
  selector:'user-form',
  templateUrl: 'user-form.component.html'
})
```

```
export class UserFormComponent {
  form: FormGroup;
  title: string;
  user = new User();
```

```
  constructor(fb: FormBuilder, private _router:Router,
    private af:AngularFire){
    this.form = fb.group({
      username:['',Validators.required ],
      email:['',Validators.required]
    })
  }
```

```
  ngOnInit(){
    this.title = "New User";
  }
```

```
  submit(){
    this.af.database.list('/').push({
      name: this.user.username,
      email: this.user.email
    });

    this._router.navigate(['']);
  }
```

```
}
```

Code Explanation

The code pertaining to generating a model driven form should be familiar to you as explained in **chapter 7: Model Driven Forms**. We will provide a brief explanation in the following sections.

```
export class UserFormComponent {  
  form: FormGroup;  
  title: string;  
  user = new User();
```

We want to remind you that it is important to initialize user to be a blank User object to avoid any null reference exception that might occur in the loading of the form either when we add a new user or later when we reuse the form again to edit an existing user.

```
  this.form = fb.group({  
    username:["",Validators.required ],  
    email:["",Validators.required]  
  })
```

We create our form using the FormBuilder object that has two controls, username and email (each having the required validator applied on it). You can of course implement and apply your own custom validators as we have gone through in chapter 7.

FirestoreListObservable Push

Here I would like to focus on the submit() method which will be called by the form upon submit.

```
submit(){  
  this.af.database.list('/').push({  
    name: this.user.username,  
    email: this.user.email  
  });  
  
  this._router.navigate([""]);  
}
```

To add an object to firestore, we use the push method from our FirestoreListObservable this.af.database.list which we covered earlier in listing users. As mentioned, FirestoreListObservable wraps the standard Observable that comes with rxjs and adds additional method such as push which we use to add a new object to

firebase.

To be able to add an object to firebase, we need to have write permission. Earlier on, we have set this to true in the firebase console.

After adding the new user, we navigate back to the list of users with `this._router.navigate([''])`.

user-form.component.html

Next, create the template of UserForm Component in `user-form.component.html` with the below codes.

```
<h1>{{ title }}</h1>
<form [formGroup]="form" (ngSubmit)="submit()">
  <div class="form-group">
    <label for="username">Username</label>
    <input [(ngModel)]="user.username" type="text" class="form-control"
      formControlName="username">
    <div *ngIf="form.controls.username.touched &&
      !form.controls.username.valid" class="alert alert-danger">
      Username is required
    </div>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input [(ngModel)]="user.email" class="form-control"
      formControlName="email">
    <div *ngIf="form.controls.email.touched && form.controls.email.errors">
      <div *ngIf="form.controls.email.errors.required"
        class="alert alert-danger">
        Email is required.
      </div>
    </div>
  </div>
  <button [disabled]="!form.valid" class="btn btn-primary" type="submit">{{ title }}</button>
</form>
```

Code Explanation

The markup for the form should be familiar to you. If not, go back to chapter 6 and 7 for a revision.

```
<label for="username">Username</label>
<input [(ngModel)]="user.username" type="text" class="form-control"
  formControlName="username">
<div *ngIf="form.controls.username.touched &&
  !form.controls.username.valid" class="alert alert-danger">
```



```
    Username is required
  </div>
```

Essentially, we have done some basic validation to the username and email fields. If no username is supplied, we display an alert message “Username is required”. If no email is supplied, we display the alert message “Email is required”. We have also applied the `form-control` class to give our form the bootstrap looking feel.

```
<button [disabled]="!form.valid" class="btn btn-primary" type="submit">{{ title }}</button>
```

We disable the **Submit** button till all fields are valid.

app.module.ts

Lastly, in `app.module.ts`, we import `ReactiveFormsModule` because we need it for model driven forms. We also import and declare `UserForm Component` to be part of `App Module`.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { AngularFireModule } from 'angularfire2';

import { AppComponent } from './app.component';
import { UserComponent } from './user.component';
import { UserFormComponent } from './user-form.component';

import { routing } from './app.routing';

export const firebaseConfig = {
  apiKey: "AIzaSyC94rD8wXG0aRLTcG29qVGw8CFfvCK7XVQ",
  authDomain: "myfirstfirebaseproject-6da6c.firebaseio.com",
  databaseURL: "https://myfirstfirebaseproject-6da6c.firebaseio.com",
  storageBucket: "myfirstfirebaseproject-6da6c.appspot.com",
  messagingSenderId: "138019512918"
};

@NgModule({
  declarations: [
    AppComponent,
    UserComponent,
    UserFormComponent,
  ],
  imports: [
    BrowserModule,
    ReactiveFormsModule,
```

```

    HttpModule,
    AngularFireModule.initializeApp(firebaseConfig),
    routing
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {

}

```

Try it yourself

As an exercise, implement dirty tracking on the form. That is, if we fill out the fields in the form and accidentally navigate away from the form page, show a confirmation box warning us that we have unsaved changes. And we click Cancel to stay on the form or Ok to navigate away. We have explained this in **chapter 10** where we covered the CanDeactivate interface.

Running your app

Run your app now. Go to the **Add** form, enter in a new username and email and upon submitting the form, you should be able to see your new user object added to the list.

12.6 Deleting a User

Next, we want to delete a user by clicking on the delete icon in a row of the user list, and a confirmation box will appear asking us if we want to delete the user.

user.component.html

First in user.component.html, we bind the click event of the delete icon to the delete() method with user object (from firebase) as argument.

```

<h1>Users</h1>
<button class="btn btn-primary" (click)="add()">Add</button>
<table class="table table-bordered">
  <thead>
    <tr>
      <th>Username</th>
      <th>Email</th>
      <th>Edit</th>
      <th>Delete</th>
    </tr>

```

```

</thead>
<tbody>
<tr *ngFor="let user of users | async">
<td>{{ user.name }}</td>
<td>{{ user.email }}</td>
<td>
    <a>
        <i class="glyphicon glyphicon-edit"></i>
    </a>
</td>
<td>
    <a>
        <i (click)="delete(user)" class="glyphicon glyphicon-remove"></i>
    </a>
</td>
</tr>
</tbody>
</table>

```

user.component.ts

In user.component.ts , we implement the delete() method as shown below.

```

import { Component, OnInit } from '@angular/core';
import { AngularFire } from 'angularfire2';
import { Router } from '@angular/router';

@Component({
  selector: 'users',
  templateUrl: './user.component.html'
})
export class UserComponent {
  users;

  constructor(private af:AngularFire, private _router: Router){
  }

  ngOnInit(){
    this.users = this.af.database.list('/');
  }

  add(){
    this._router.navigate(['add']);
  }

  delete(user){
    if (confirm("Are you sure you want to delete " + user.name + "?")){

```

```

    this.af.database.object(user.$key).remove()
      .then( x=> console.log("SUCCESS"))
      .catch( error => {
        alert("Could not delete the user.");
        console.log("ERROR", error)
      });
  }
}
}

```

In the delete() method, we first display a confirmation box asking for confirmation to delete. If true, we call the remove() method of this.af.database.object .

```

this.af.database.object(user.$key).remove()

```

The object() method allows us to get one single specific object from firebase. We need to specify the location of the data in firebase as argument in object() . In this case, the location of the object is contained in the \$key property of the user object we have clicked to delete. How did we get this \$key property? Whenever we add an object to firebase, a unique key is generated for us. We use this unique key stored in \$key to retrieve the object for deletion, and also later for retrieval and update.

Having specified the targeted object using object() , we call the remove() method to remove it from firebase. remove() returns a promise which you can optionally subscribe to be notified if the deletion is successful. (the same applies for push() and update())

```

this.af.database.object(user.$key).remove()
  .then( x=> console.log("SUCCESS"))
  .catch( error => {
    alert("Could not delete the user.");
    console.log("ERROR", error)
  });

```

If successful, we log “Success”, and if an error is caught, we log a error message.

12.7 Populating the Form on Edit

Having implemented, list, add and delete, we will now implement edit. But before we can implement edit, we need to retrieve the existing requested user object and populate it on the form first. When a user clicks on the **Edit** icon, she would be navigated to the User Form with the given user details populated in the input fields. We also change the title of the page to **Edit User** instead of **Add User**. And if we access the User Form via the Add User button, title should be **New User**.

First in app.routing.ts , we define a new route add/:id with id being a parameter as

shown below. id will contain our user object id used to retrieve our user object and populate the Edit form.

app.routing.ts

```
import { RouterModule } from '@angular/router';
import { UserComponent } from './user.component';
import { UserFormComponent } from './user-form.component';

export const routing = RouterModule.forRoot([
  { path:'', component:UserComponent },
  { path:'add',component:UserFormComponent },
  { path:'add/:id', component: UserFormComponent }
]);
```

user.component.html

Next, in user.component.html , we add the router link to the **Edit**icon with the parameter user.\$key used to retrieve our user object and populate our form.

```
<a [routerLink]="['/add', user.$key]">
  <i class="glyphicon glyphicon-edit"></i>
</a>
```

user-form.component.ts

Next in user-form.component.ts , add the codes below in **bold**.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router, ActivatedRoute } from '@angular/router';
import { AngularFire } from 'angularfire2';

import { User } from './user';

@Component({
  selector:'user-form',
  templateUrl: 'user-form.component.html'
})
export class UserFormComponent {
  id;
  form: FormGroup;
  title: string;
  user = new User();
  item;
```

```

    constructor(fb: FormBuilder, private _router:Router, private _route:ActivatedRoute, private
af:AngularFire){
    this.form = fb.group({
        username:['',Validators.required ],
        email:['',Validators.required]
    })
}

ngOnInit(){
    this._route.params.subscribe(params => {
        this.id = params["id"];
    });

    if(!this.id){
        this.title = "New User";
    }
    else{
        this.title = "Edit User";
        this.item = this.af.database.object(this.id);
    }
}

submit(){
    this.af.database.list('/').push({
        name: this.user.username,
        email: this.user.email
    });

    this._router.navigate(['']);
}
}

```

Code Explanation

import { Router, **ActivatedRoute** } from '@angular/router';

We import **ActivatedRoute** and inject it in our constructor. This is used to retrieve the parameter **id** passed in from User component as shown below.

```

ngOnInit(){
    this._route.params.subscribe(params => {
        this.id = params["id"];
    });

    if(!this.id){
        this.title = "New User";
    }
    else{

```

```

    this.title = "Edit User";
    this.item = this.af.database.object(this.id);
  }
}

```

We retrieve `id` from `_route.params` and if it is null, it means that we arrive at `UserForm` without a parameter and want to perform adding a new user. We thus set the title to “New User”.

If `id` is not null, it means we want to edit a user of that given `id` and therefore display title as “Edit User”. We then proceed to retrieve the user object with the below code:

```

this.item = this.af.database.object(this.id);

```

After retrieving our user object with the `object` method, we assign it to `item` variable. With `item` now containing our requested user object, we can populate our edit form.

user-form.component.html

In `user-form.component.html` , add the below two portions of code in **bold**.

```

<h1>{{ title }}</h1>
<form [formGroup]="form" (ngSubmit)="submit()">
  <div class="form-group">
    <label for="username">Username</label>
    <input [(ngModel)]="user.username" type="text" class="form-control"
      formControlName="username" value={{(item|async)?.name}}>
    <div *ngIf="form.controls.username.touched &&
      !form.controls.username.valid" class="alert alert-danger">
      Username is required
    </div>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input [(ngModel)]="user.email" class="form-control"
      formControlName="email" value={{(item|async)?.email}}>
    <div *ngIf="form.controls.email.touched && form.controls.email.errors">
      <div *ngIf="form.controls.email.errors.required"
        class="alert alert-danger">
        Email is required.
      </div>
    </div>
  </div>
  <button [disabled]="!form.valid" class="btn btn-primary" type="submit">{{ title }}</button>
</form>

```

The code above populates the `value` property of the username and email input fields from the `item` object. We use string interpolation `{{(item|async)?.email}}` to render the

values. As mentioned earlier, we use the async pipe to subscribe to item which is a `FirebaseObjectObservable` to check for the latest value emitted.

After applying the async pipe, we need to apply the elvis operator ? which means we access the email property only when `item|async` is not null. This is because `item|async` is created dynamically at runtime and can be initially null as there is a bit of delay from the moment we subscribe till the moment we get the result from firebase.

12.8 Updating a User

Finally to update the user, we make some code changes and additions to `submit()` in `user-form.component.ts` . Fill in the below code into `submit()` .

user-form.component.ts

```
submit(){
  if (this.id) {
    this.af.database.object(this.id).update({
      name: this.user.username,
      email: this.user.email
    });
  }
  else{
    this.af.database.list('/').push({
      name: this.user.username,
      email: this.user.email
    });
  }

  this._router.navigate([""]);
}
```

Code Explanation

We first check if there is an id , which means the form is in edit mode. If so, we call the update method of object to update . Else, which means the form is in Add New User mode, we call `push()` of object to add the new user object to firebase.

Running your App

If you run your app now, your app should have full functionality to create, update, delete and read user data from and to firebase.

Summary

In this chapter, we learnt how to implement C.R.U.D. operations using Firebase as our backend. We learnt how to add firebase to our application, how to work with the firebase database from the firebase console, how to display a list of users, how to add a user with the push method, how to delete a user with the remove method, retrieve a single firebase object to prepare our form for edit and how to update a user.

With this knowledge, you can move on and build more complicated enterprise level fully functional Angular applications of your own!

Hopefully you've enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve. Please feel free to email me at support@i-ducate.com. Contact me also if you have not already to have the full source code for this chapter or if you encounter any errors with your code.

If you didn't like the book, please email us and let us know how we could improve it. This book can only get better thanks to readers like you.

If you like the book, I would appreciate if you could leave us a review too.

Thank you and all the best to your learning journey in Angular!

ABOUT THE AUTHOR

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Outside the programming world, Greg is happily married to his wife, a proud father of three boys and greatly involved in church work. Contact Greg at support@i-ducate.com.