Shravan Kumar Kasagoni

Building Modern Web Applications Using Angular

Learn how to create rich and compelling web applications with Angular

Packt>

Building Modern Web Applications Using Angular

Learn how to create rich and compelling web applications with Angular

Shravan Kumar Kasagoni

Packt>

BIRMINGHAM - MUMBAI

Building Modern Web Applications Using Angular

Credits

Author
Shravan Kumar Kasagoni

Copy Editors
Safis Editing
Dipti Mankame

Reviewers
Hemant Singh
Phodal Huang

Project Coordinator
Judie Jose

Acquisition Editors
Tushar Gupta

Proofreader
Safis Editing

Content Development Editor
Juliana Nair

Indexer
Rekha Nair

Technical Editor
Mohd Riyan Khan

Graphics
Kirk D'Penha

Production Coordinator
Melwyn Dsa

About the Author

Shravan Kumar Kasagoni is a developer, gadget freak, technology evangelist, mentor, blogger, and speaker living in Hyderabad. He has been passionate about computers and technology right from childhood. He holds a bachelors degree in computer science and engineering, and he is a Microsoft Certified Professional.

His expertise includes modern web technologies (HTML5, JavaScript, and Node.js) and frameworks (Angular, React.js, Knockout.js, and so on). He has also worked on many Microsoft technologies, such as ASP.NET MVC, ASP.NET WEB API, WCF, C#, SSRS, and the Microsoft cloud platform Azure.

He is a core member of Microsoft User Group Hyderabad, where he helps thousands of developers on modern web technologies and Microsoft technologies. He also actively contributes to open source community. He is a regular speaker at local user groups and conferences. Shravan has been awarded Microsoft's prestigious Most Valuable Professional award for past 6 years in a row for his expertise and community contributions in modern web technologies using ASP.NET and open source technologies.

He is currently working with Novartis India, where he is responsible for the design and development of high-performance modern enterprise web applications and RESTful APIs. Previously, he was associated with Thomson Reuters and Pramati Technologies.

About the Reviewers

Hemant Singh is a developer living in Hyderabad/AP, India. Currently, he is working for Microsoft as a UX consultant. He loves open source, and is active in various projects. Hemant is not much of a blogger, but tries to share information whenever possible.

He handcrafts CSS and HTML documents and handles JavaScript (the good parts). It won't be surprising if he tells you that he fell in love with HTML5 and, of course, CSS3. Hemant also has a passion for user interface and experience design and tries to show some of his work in his portfolio.

Phodal Huang is a developer, creator, and author. He works for ThoughtWorks as a consultant. Currently, he focuses on IoT and frontend development. He is the author of Design Internet of Things and Growth: Thinking in Full Stack in Chinese .

He is an open source enthusiast, and has created a series of projects in GitHub. After his daily work, he likes to reinvent some wheels for fun. He created the application Growth with Ionic 2 and Angular 2 , which is about coaching newbies about programming. You can find out more about wheels on his GitHub page, `http://github.com/phodal`.

He loves designing, writing, hacking, and traveling. You can also find out more about him on his personal website at `http://www.phodal.com`.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1785880721`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Building Modern Web Applications Using Angular helps readers to design and develop modern web applications. It provides a solid understanding of the Angular 4 framework. Readers will learn how to build and architect high-performance web applications mainly focusing on UI. This is an end-to-end guide for all the new features in Angular 4. This book also covers some of the latest JavaScript concepts in ECMAScript 2015, ECMAScript 2016, and TypeScript.

This book will take you from nowhere when it comes to building UI applications for Web and mobile to become a master using Angular 4. It will explain almost every feature of the Angular 4 framework with a particle approach and lots of examples, showing how to use them in real-world scenarios to build compelling UI applications. Chapters at the end of the book are dedicated to show how to build an end-to-end application UI using individual Angular 4 features that were explained in previous chapters.

What this book covers

`Chapter 1`, *Getting Started*, introduces the Angular 4 framework and its new features, how Angular 4 is better and more powerful than its predecessor Angular 1, development environment setup for Angular 4 applications. Also, it provides a quick insight into TypeScript and its features, how to write a basic Angular 4 app, and understanding its anatomy.

`Chapter 2`, *Basics of Components*, walks through the basics of Angular 4 components, starting with display data using interpolation syntax, property binding, attribute binding, working with DOM events, and two-way data binding. It also introduces structural directives for conditionally displaying data and attribute directives for conditional styling.

`Chapter 3`, *Components, Services, and Dependency Injection*, walks through how to develop Angular 4 applications using multiple components, communicating between components, sharing the data between these components using services and injecting services using dependency injection, as well as how dependency injection works.

`Chapter 4`, *Working with Observables*, focuses on reactive programming, Observables, RxJS, how Angular 4 implements Observables using RxJS, and using Observables and RxJS operators in Angular 4 applications.

`Chapter 5`, *Handling Forms*, introduces how to create different types of forms in Angular 4 applications to accept user input and validate it using template-driven forms, reactive forms, and validation directives.

`Chapter 6`, *Building a Book Store Application*, walks through how to structure tiny to a complex application using Angular 4 modules, build various types of user interfaces in a Book Store application, navigate between components using routing, interact with the server using the HTTP service.

`Chapter 7`, *Testing*, introduces how to write unit tests using Jasmine and Angular Test Utilities for various parts of Angular 4 applications.

`Chapter 8`, *Angular Material*, walks through how to build a single compelling UI, which flows across desktops, tablets, and mobile devices, using material design, and learning how to customize material design as per customer branding.

What you need for this book

This book assumes a basic knowledge of JavaScript, web development, how to use command line, Git, and **node package manager** (**npm**).

In this book, you will need the following software list:

- Operating system:
  - MAC OS X 10.9 and higher
  - WINDOWS 7 and higher
- Node.js 6:
  - **MAC**: `https://nodejs.org/dist/v6.10.3/node-v6.10.3.pkg`
  - **Windows**: `https://nodejs.org/dist/v6.10.3/node-v6.10.3-x64.msi`
- Any code editor
- Visual Studio Code
- Sublime

Internet connectivity is required to install the necessary npm packages.

Who this book is for

This book is for developers building web applications who are new to the Angular world and interested in creating modern, responsive, complex UI applications using Angular 4. For developers who are already working with the AngularJS 1 framework, this book provides an upgrading path with new concepts.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, folder names, filenames, file extensions, path names, URLs, user input, and Twitter handles are shown as follows:

A block of code is set as follows:

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

@Component({
  selector: 'hello-world-app',
  template: '<h1>Say Hello to Angular</h1>'
})
class HelloWorldAppComponent { }
```

Any command-line input or output is written as follows:

```
$ npm install json-server -save
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The complete set of code can also be downloaded from the following GitHub repository: `ht tps://github.com/PacktPublishing/Building-Modern-Web-Application-using-Angul ar`. We also have other code bundles from our rich catalog of books and videos available at: `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1

Getting Started

In this chapter, we are going to learn the Angular framework basics and new features. We will learn how to use features from future versions of JavaScript and TypeScript to develop modern web applications using Angular. After going through this chapter, the reader will understand the following concepts:

- The Angular framework and its new features
- Setting up the environment for development
- The basics of TypeScript
- Angular application basics
- How to write your first Angular application?

Introduction to Angular

What is Angular? It is an open source modern JavaScript framework in order to build the Web, mobile web, native mobile, and native desktop applications. It can also be used in combination with any server-side web application framework, such as ASP.NET and Node.js. Angular is the successor of AngularJS 1, which is one of the best JavaScript frameworks for building client-slide web applications, used by millions of developers across the globe.

Even though the developer community highly appreciates AngularJS 1, it has its challenges. Many developers feel that the AngularJS 1 learning curve is high. Out-of-the-box performance in complex applications with a huge amount of data is not that great, which is essential in enterprise applications. The API to build directives is very confusing and complex, even for an experienced AngularJS 1 programmer, which is the key concept to build UI applications based component-based architecture.

Angular is the next version of AngularJS 1.x, but it is a complete rewrite from its predecessor. It is built on top of the latest web standards (web components, Observables, and decorators), the learning curve is minimal, and performance is better. This book will cover the latest version of Angular, which is version 4, at the time of writing this book. Angular 4 is an incremental update from Angular 2, unlike Angular 2.

> When we use the term Angular, we are referring to the latest version of the framework. Anywhere we need to discuss version 1.x, we will use the term AngularJS.

What is new in Angular?

Many concepts in version 1.x are irrelevant in Angular, such as `controller`, directive definition object, jqLite, `$scope`, and `$watch`. Even though lots of concepts are irrelevant, lots of goodness in AngularJS 1.x is carried forward to Angular such as services, dependency injection, and pipes. Here is a list of the new features in Angular:

- Components
- New templating syntax
- Unidirectional data flow
- Ultra-fast change detection
- New component router
- Observables
- Server-side rendering
- New languages for development
- ES2015
- TypeScript
- Ahead of time compilation

Setting up the environment

We can start developing our Angular applications without any setup. However, we are going to use Node.js and **npm** (**node package manager**) for tooling purposes. We need them for downloading tools, libraries, and packages. We also use them for build process automation. Angular applications are as follows:

- Node.js is a platform built on top of V8, Google's JavaScript runtime, which also powers the Chrome browser
- Node.js is used for developing server-side JavaScript applications
- The npm is a package manager for Node.js, which makes it quite simple to install additional tools via packages; it comes bundled with Node.js
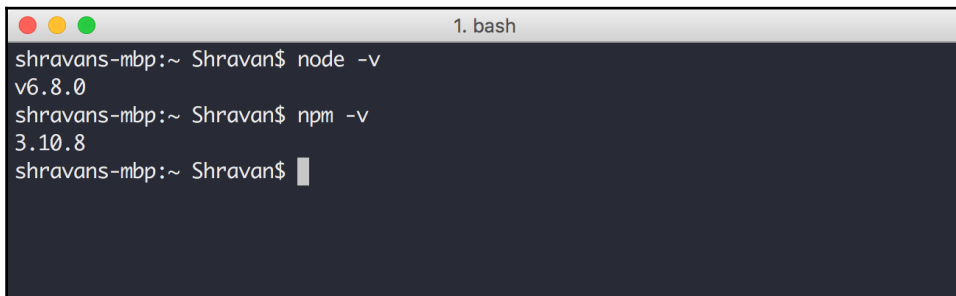
Installing Node.js and npm

The easiest way to install Node is to follow the instructions at: `https://nodejs.org`. Download the latest version of Node for the respective operating system and install it. The npm is installed as part of the Node.js installation.

Once we install Node.js, run the following commands on the command line in Windows or Terminal in macOS to verify that Node.js and npm are installed and set up properly:

```
$ node -v
$ npm -v
```

The preceding commands will display currently installed Node.js and npm versions, respectively:

> The Angular documentation recommends installing at least Node.js v4.x.x, NPM 3.x.x, or higher versions.

Language choices

We can write the Angular applications in JavaScript or any language that can be compiled into JavaScript. Here, we are going to discuss the three primary language choices.

[8]

ECMAScript 5

ECMAScript 5 (**ES5**) is the version of JavaScript that runs in today's browsers. The code written in ES5 does not require any additional transpilation or compilation, and there is no new learning curve. We can use this as it is to develop Angular applications.

ECMAScript 2015

ECMAScript 2015 (**ES2015**), previously known as **ECMAScript 6** (**ES6**), is the next version of JavaScript. ES2015 is a significant update to the JavaScript language; the code written in ES2015 is a lot cleaner than ES5. ES2015 includes a lot of new features to increase the expressiveness of JavaScript code (for example, classes, arrows, and template strings).

However, today's browsers do not support all the features of ES2015 completely (for more information please refer to: `https://caniuse.com/#search=es6`). We need to use transpilers like **Babel** to transform ES2015 code into ES5-compatible code so that it works in today's browsers.

> Babel is a JavaScript transpiler, and it transpiles our code written in ES2015 to ES5 that runs in our browsers (or on your server) today. Learn more about Babel at `https://babeljs.io`.

TypeScript

TypeScript is a superset of JavaScript, which means all the code written in JavaScript is valid TypeScript code, and TypeScript compiles back to simple standards-based JavaScript code, which runs on any browser, for any host, on any OS. TypeScript lets us write idiomatic JavaScript. It provides optional static types that are useful for building scalable JavaScript applications and other features such as classes, modules, and decorators from future versions of the JavaScript.

It provides the following features to improve developer productivity and build scalable JavaScript applications:

- Static checking
- Symbol-based navigation
- Statement completion
- Code refactoring

These features are critical in large-scale JavaScript application developments, so in this book, we are going to use TypeScript for writing Angular applications. The Angular framework itself was developed in TypeScript.

> All the ECMAScript 5 code is valid in ECMAScript 2015; all ECMAScript 2015 code is valid TypeScript code.

Installing TypeScript

We can install TypeScript in multiple ways, and we are going to use npm, which we installed earlier. Go to the command line in Windows or Terminal in macOS and run the following command:

```
$ npm install -g typescript
```


The preceding command will install the TypeScript compiler and makes it available globally.

TypeScript basics – types

TypeScript uses `.ts` as a file extension. We can compile TypeScript code into JavaScript by invoking the TypeScript compiler using the following command:

```
$ tsc <filename.ts>
```

JavaScript is a loosely typed language, and we do not need to declare the type of a variable ahead of time. The type will be determined automatically while the program is being executed. Because of the dynamic nature of JavaScript, it does not guarantee any type safety. TypeScript provides an optional type system to ensure type safety.

In this section, we are going learn the important data types in TypeScript.

String

In TypeScript, we can use either double quotes (") or single quotes (') to surround strings similar to JavaScript.

```
var bookName: string = "Angular";
bookName = 'Angular UI Development';
```

Number

As in JavaScript, all numbers in TypeScript are floating point values:

```
var version: number = 4;
```

Boolean

The `boolean` data type represents the `true/false` value:

```
var isCompleted: boolean = false;
```

Array

We have two different syntaxes to describe arrays, and the first syntax uses the element type followed by `[]`:

```
var fw: string[] = ['Angular', 'React', 'Ember'];
```

The second syntax uses a generic array type, `Array<elementType>`:

```
var fw: Array<string> = ['Angular', 'React', 'Ember'];
```

Enum

TypeScript includes the `enum` data type along with the standard set of data types from JavaScript. In languages such as C# and JAVA, an `enum` is a way of giving more friendly names to sets of numeric values, as shown in the following code snippet:

```
enum Frameworks { Angular, React, Ember };
var f: Frameworks = Frameworks.Angular;
```

The numbering of members in the enum type starts with 0 by default. We can change this by manually setting the value of one of its members. As illustrated earlier, in code snippet, the `Frameworks.Angular` value is 0, the `Frameworks.React` value is 1, and the `Frameworks.Ember` value is 2.

We can change the starting value of enum type to 5 instead of 0, and remaining members follow the starting value:

```
enum Frameworks { Angular = 5, React, Ember };
var f: Frameworks = Frameworks.Angular;
var r: Frameworks = Frameworks.React;
```

In the preceding code snippet, the `Frameworks.Angular` value is 5, the `Frameworks.React` value is 6, and the `Frameworks.Ember` value is 7.

Any

If we need to opt out type-checking in TypeScript to store any value in a variable whose type is not known right away, we can use any keyword to declare that variable:

```
var eventId: any = 7890;
eventId = 'event1';
```

In the preceding code snippet while declaring a variable, we are initializing it with the number value, but later we are assigning the string value to the same variable. TypeScript compiler will not report any errors because of any keyword. Here is one more example of an array storing different data type values:

```
var myCollection:any[] = ["value1", 100, 'test', true];
myCollection[2] = false;
```

Void

The void keyword represents not having any data type. Functions without return keyword do not return any value, and we use void to represent it.

```
function simpleMessage(): void {
    alert("Hey! I return void");
}
```

Let's write our first TypeScript example and save it into the `example1.ts` file:

```
var bookName: string = 'Angular UI Development';
var version: number = 2;
var isCompleted: boolean = false;
var frameworks1: string[] = ['Angular', 'React', 'Ember'];
var frameworks2: Array<string> = ['Angular', 'React', 'Ember'];
enum Framework { Angular, React, Ember };
var f: Framework = Framework.Angular;
var eventId: any = 7890;
eventId = 'event1';
var myCollection:any[] = ['value1', 100, 'test', true];
myCollection[2] = false;
```

Let's compile the `example1.ts` file using the TypeScript command-line compiler:

```
$ tsc example1.ts
```

The preceding command will compile TypeScript code into plain JavaScript code into the `example1.js` file; this is how it will look, and it is plain JavaScript:

```
var bookName = 'Angular UI Development';
var version = 2;
var isCompleted = false;
var frameworks1 = ['Angular', 'React', 'Ember'];
var frameworks2 = ['Angular', 'React', 'Ember'];
var Framework;
(function (Framework) {
    Framework[Framework["Angular"] = 0] = "Angular";
    Framework[Framework["React"] = 1] = "React";
    Framework[Framework["Ember"] = 2] = "Ember";
})(Framework || (Framework = {}));
;
var f = Framework.Angular;
var eventId = 7890;
eventId = 'event1';
var myCollection = ['value1', 100, 'test', true];
myCollection[2] = false;
```

Functions

Functions are the fundamental building blocks of any JavaScript application. In JavaScript, we declare functions in two ways.

Function declaration – named function

The following is an example of `function` declaration:

```
function sum(a, b) {
return a + b;
}
```

Function expression – anonymous function

The following is an example of `function` expression:

```
var result = function(a, b) {
    return a + b;
}
```

In JavaScript, unlike any other concept, there is no type safety for functions also. We do not have any assurance on data types of parameters, return type, the number of parameters passed to function, TypeScript guarantees all this. It supports both the syntaxes.

Here are the same functions written in TypeScript:

```
function sum(a: number, b: number): number {
    return a + b;
}

var result = function(a: number, b: number): number {
return a + b;
}
```

The preceding TypeScript functions are very similar to the JavaScript syntax except parameters, and return type has type on them, which ensures type safety while invoking them.

Classes

The ES5 does not have the concept of classes. However, we can mimic the class structure using different JavaScript patterns. The ES2015 does support classes. However, today, we can already write them in TypeScript. In fact, the ECMAScript 2015 class syntax is inspired by TypeScript.

The following example shows a person class written in TypeScript:

```
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHello() {
        return 'Hello ' + this.name;
    }
}

var person = new Person('Shravan');
console.log(person.name);
console.log(person.sayHello());
```

The preceding `Person` class has three members - a property named `name`, a `constructor`, and a method `sayHello`. We should use this keyword to refer to the properties of the class. We created an instance of the `Person` class using the new operator. In the next step, we invoke the `sayHello()` method of the `Person` class using its instance created in the previous step.

Save the preceding code into the `person.ts` file and compile it using the TypeScript command-line compiler. It will compile TypeScript code into plain JavaScript code into the `person.js` file:

```
$ tsc person.ts
```

Here is the plain JavaScript code, which was compiled from the TypeScript class:

```
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    Person.prototype.sayHello = function () {
        return 'Hello ' + this.name;
    };
    return Person;
}());

var person = new Person('Shravan');
console.log(person.name);
console.log(person.sayHello());
```

> To learn more about functions, classes, and other concepts in TypeScript, check out `http://typescriptlang.org`.

Writing your first Angular application

Angular follows a component-based approach to building an application. An application written in AngularJS 1 is a set of individual controllers and views, but in Angular, we need to treat our application as a component tree.

The Angular application component tree will have one root component; it will act as the entry point of the application. All the other components that are part of the application will load inside the root component, and they can be nested in any way that we need inside the root component.

Angular also has the concept of modules, which are used for grouping components with similar functionality. An Angular application should have minimum one module and minimum one component that should be part of that module. Component acts as root component, and module acts as root module.

Set up the Angular application

Let' us begin writing our first Angular application by creating the following folder structure and files:

```
hello-world
├── index.html
├── package.json
├── tsconfig.json
└── src
└── app.ts
```

Step 1

As we are going to write our application in TypeScript, let us begin with the `tsconfig.json` file first. It is the TypeScript configuration file that contains instructions for its compiler on how to compile TypeScript code into JavaScript. If we do not use the `tsconfig.json` file, the TypeScript compiler uses the default flags during compilation, or we can pass our flags manually every time while compiling.

The `tsconfig.json` file is the best way to pass the flags to the TypeScript compiler, so we do not need to type them every time. Some of the flags here are mandatory for the Angular application written in TypeScript; we are going to use this file throughout the book. Add the following code to the `tsconfig.json` file:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": ["es2015", "dom"],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true,
    "typeRoots": ["node_modules/@types/"]
  },
  "compileOnSave": true,
  "exclude": ["node_modules/*"]
}
```

> **Downloading the example code**
> Detailed steps to download the code bundle are mentioned in the Preface of this book. The code bundle for the book is also hosted on GitHub at: `https://github.com/PacktPublishing/Building-Modern-Web-Application-using-Angular`. We also have other code bundles from our rich catalog of books and videos available at: `https://github.com/PacktPublishing/`. Check them out!

The explanation for flags specified in the `tsconfig.json` file are as follows:

- `target`: Specifies ECMAScript target version: `es3` (default), `es5`, or `es2015`
- `module`: Specifies module code generation: `commonjs`, `amd`, `system`, `umd` or `es2015`
- `moduleResolution`: Specifies module resolution strategy: `node` (Node.js) or `classic` (TypeScript pre-1.6)
- `sourceMap`: If `true`, generates corresponding `.map` file for `.js` file
- `emitDecoratorMetadata`: If `true`, enables the output JavaScript to create the metadata for the decorators
- `experimentalDecorators`: If `true`, enables experimental support for ES7 decorators

- `lib`: Specifies library files to be included in the compilation
- `noImplicitAny`: If `true`, raise error if we use `any` type on expressions and declarations

Step 2

Let us add the following code to the `package.json` file, which holds metadata for npm and contains all the Angular packages and third-party libraries required for Angular application development:

```json
{
  "name": "hello-world",
  "version": "1.0.0",
  "scripts": {
    "prestart": "npm run build",
    "start": "concurrently \"tsc -w\"  \"lite-server\"",
    "build": "tsc"
  },
  "license": "ISC",
  "dependencies": {
    "@angular/common": "^4.0.0",
    "@angular/compiler": "^4.0.0",
    "@angular/core": "^4.0.0",
    "@angular/platform-browser": "^4.0.0",
    "@angular/platform-browser-dynamic": "^4.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
    "systemjs": "0.20.12",
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "@types/node": "^6.0.45",
    "concurrently": "^3.4.0",
    "lite-server": "^2.3.0",
    "typescript": "~2.2.0"
  }
}
```

In the preceding code snippet, there are two important sections:

- `dependencies`: This holds all packages required for application to run
- `devDependencies`: This holds all packages required only for development

---

[18]

Once we add the preceding code to the `package.json` file in our project, we should run the following command at the root of our application:

```
$ npm install
```

The preceding command will create the `node_modules` folder in the root of project, and it downloads all the packages mentioned in the `dependencies` and `devDependencies` sections into the `node_modules` folder.

There is one more section in the `package.json` file, that is, scripts. We will discuss the scripts section when we are ready to run our application.

Step 4

We have the basic setup ready for our application; now let's write some code by beginning with a module. An Angular module in TypeScript is simply a class annotated it with the `@NgModule()` decorator.

Add the following code to the `app.ts` file under the `src` folder:

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

@NgModule({
  imports: [BrowserModule],
  declarations: [],
  bootstrap: []
})
class HelloWorldAppModule {}
```

We imported the `NgModule` decorator from the `@angular/core` module and `BrowserModule` from the `@angular/platform-browser` module using ES2015 module `imports` syntax; we will discuss these modules later.

We declared a class and annotated it with the `@NgModule()` decorator. The `@NgModule()` decorator takes a configuration object as the parameter with a couple of properties; here is what they mean.

imports	We need to specify the other modules on which our module is dependent. We are going to run our application in a browser, so our module depends on `BrowserModule`; we imported it and added it to this array.
declarations	We need to specify that the components, directives, and pipes belong to this module.
bootstrap	We need to specify the components that must be bootstrapped when this module is bootstrapped. The components added here will automatically be added to the `entryComponents` property. The components specified in the `entryComponents` will be compiled when the module is defined.

The `@NgModule()` is a decorator; the decorator is a function that adds the metadata to class declaratively without modifying its original behavior.

Step 5

In the previous step, we created a module, but the module does not do anything. It is just a container for components; actual logic needs to be written inside a component. Let's write our first component in Angular. An Angular component in TypeScript is simply a class annotated with the `@Component()` decorator:

```
@Component({})
class HelloWorldAppComponent {}
```

The `@Component()` decorator tells Angular that this class is an Angular component, and we can pass a configuration object to the `@Component()` function that has two properties - a selector and a template:

```
@Component({
  selector: 'hello-world-app',
  template: '<h1>Say Hello to Angular</h1>'
})
```

The `selector` property specifies a CSS selector (custom tag name) for the component that can be used in HTML.

The `template` property specifies the HTML template for the component that tells Angular how to render a view. Our template is a single line of HTML `Say Hello to Angular` surrounded with the `h1` tag. We can also specify a multiline template string. Instead of using the inline template, we can use the external template stored in a different file using the `templateUrl` property.

Let us also add this code to the `app.ts` file under the `src` folder:

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

@Component({
  selector: 'hello-world-app',
  template: '<h1>Say Hello to Angular</h1>'
})
class HelloWorldAppComponent {}
```

Step 6

We have our component ready, and we need to associate this component to a module. Let's add the component to the `declarations` array of the app module created in *Step 4*, and we also need this component bootstrapped as soon as a module is bootstrapped, so add this to the `bootstrap` array also. Let us add all this code to the `app.ts` file under the `src` folder:

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

@Component({
  selector: 'hello-world-app',
  template: '<h1>Say Hello to Angular</h1>'
})
class HelloWorldAppComponent {}

@NgModule({
  imports: [BrowserModule],
  declarations: [HelloWorldAppComponent],
  bootstrap: [HelloWorldAppComponent]
})
class HelloWorldAppModule {}
```

Step 7

The next step is to bootstrap our module. This can be done using the `bootstrapModule()` method; it is available in the `PlatformRef` class. We can get the instance of the `PlatformRef` class using the `platformBrowserDynamic()` function available in the `@angular/platform-browser-dynamic` module:

```
import { platformBrowserDynamic } from
                        '@angular/platform-browser-dynamic';

platformBrowserDynamic().bootstrapModule(HelloWorldAppModule);
```

The `app.ts` file finally looks as follows after adding the module bootstrapping logic:

```
import { NgModule, Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from
                        '@angular/platform-browser-dynamic';

@Component({
  selector: 'hello-world-app',
  template: '<h1>Say Hello to Angular</h1>'
})
class HelloWorldAppComponent { }

@NgModule({
  imports: [BrowserModule],
  declarations: [HelloWorldAppComponent],
  bootstrap: [HelloWorldAppComponent]
})
class HelloWorldAppModule { }

platformBrowserDynamic().bootstrapModule(HelloWorldAppModule);
```

Step 8

Let us use the component we created in the earlier step, add following code in `index.html`:

```
<html>
<head>
  <title>Angular Hello World</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
                                    initial-scale=1">
```

```
    <script src="node_modules/core-js/client/shim.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script>
      System.config({
        paths: {
          'ng:': 'node_modules/@angular/'
        },
        map: {
          '@angular/core': 'ng:core/bundles/core.umd.js',
          '@angular/common': 'ng:common/bundles/common.umd.js',
          '@angular/compiler': 'ng:compiler/bundles/compiler.umd.js',
          '@angular/platform-browser': 'ng:platform-
                            browser/bundles/platform-browser.umd.js',
          '@angular/platform-browser-dynamic': 'ng:platform-browser-
                       dynamic/bundles/platform-browser-dynamic.umd.js',
          'rxjs': 'node_modules/rxjs'
        },
        packages: {
          src: {
            main: 'app',
            defaultExtension: 'js'
          },
          rxjs: { defaultExtension: 'js' }
        }
      });

      System.import('src').catch(function (err) {
        console.error(err);
      });
    </script>
</head>
<body>
<hello-world-app>Loading...</hello-world-app>
</body>
</html>
```

There is a lot going on in the `index.html`; let us understand it step by step.

SystemJS

We included SystemJS and its configuration, so what is SystemJS? SystemJS is a universal dynamic module loader. It can load ES2015, AMD, CommonJS modules, and global scripts in the browser and Node.js.

```
<script src="node_modules/systemjs/dist/system.src.js"></script>

<script>
System.config({});
System.import('src');
</script>
```

Why do we need SystemJS? If we refer back to our previous steps, we imported `NgModule`, `Component` from the `@angular/core` module, `BrowserModule` from the `@angular/platform-browser` module, the `platformBrowserDynamic` function from the `@angular/platform-browser-dynamic` module. The `@angular/core` module is a physical JavaScript file available in our application root at the `node_modules/@angular/core/bundles/core.umd.js` path.

Normally, in order to use anything (variables, functions, objects, and so on) in an external JavaScript file, we need to add it to our HTML using the `<script>` tag. Instead of using the traditional `<script>` tag to load our JavaScript files, which is slow and synchronous, we can use the ES2015 modules feature to load our JavaScript files dynamically, asynchronously, and on demand. Module loading can be done using SystemJS, and instead of SystemJS, we can also use other alternatives such as webpack and rollup.

We need to tell the SystemJS how to load modules using its configuration. In SystemJS configuration, map object tells where to look for JavaScript files, packages object tells how to load when no filename is specified and no file extension is specified.

After the configuration, we need inform one more thing to SystemJS that is our main file, which system loads it first and start the execution. This is done using the `System.import()` method.

In our code, we specified `System.import('src')`. SystemJS looks for the `src` folder in configuration; we specified the `app.js` file is the main file for it. SystemJS will load the `app.js` file under `src` and start the execution. Currently, we have only one JavaScript file with all the logic in our application, `app.js`.

Using the Angular component

In the `body` tag section, we are using the `<hello-world-app>` tag, which we declared in our `HelloWorldAppComponent`, and it will render its view:

```
<body>
  <hello-world-app>Loading...</hello-world-app>
</body>
```

Understanding npm packages

Let us refer to *Step 3*; in the `package.json` file `dependencies` section, we have a lot of packages. We downloaded them, and some of them are loaded using the `<script>` tag, while some of them are loaded using SystemJS. Some of these packages are part of the Angular framework; some are third-party libraries.

The following are the Angular packages:

- `@angular/core`: This package contains critical runtime parts of the angular framework required by every application. It includes all metadata decorators, dependency injection, `NgModule`, `Component`, directive, the component life cycle hooks, core providers, links to change detection and Observables.
- `@angular/common`: This package contains common directives (`ngClass`, `ngFor`, `ngIf`, `ngPlural`, `ngPluralCase`, `ngStyle`, `ngSwitch`, `ngSwitchCase`, `ngSwitchDefault`, and `ngTemplateOutlet`), pipes (`AsyncPipe`, `DatePipe`, `I18nPluralPipe`, `I18nSelectPipe`, `JsonPipe`, `LowerCasePipe`, `CurrencyPipe`, `DecimalPipe`, `PercentPipe`, `SlicePipe`, and `UpperCasePipe`), and services.
- `@angular/compiler`: This package contains logic for the Angular template compiler.
- `@angular/platform-browser`: This package contains everything related to DOM and browser.
- `@angular/platform-browser-dynamic`: This package contains everything related to bootstrapping our applications during development.

The following are the `dependencies`:

- `rxjs`: This is a reactive extensions library for JavaScript based on Observables, which are used by Angular. JavaScript native does not support them, and there is a proposal for adding them as core language in future versions. However, we need to include RxJS and its mandatory dependency. We are loading it using the SystemJS configuration.
- `zone.js`: This library contains all the logic for change detection; there is a proposal for adding them as core language. However, we need to include `zone.js` and its mandatory dependency. We load it using the `<script>` tag.

> `@angular/core` package is dependent on `rxjs`, `zone.js`.

- `core-js`: It is a polyfill for ECMAScript 2015 and the upcoming JavaScript language features. We need this for Angular to work in older IE browsers where ever it is supported, we load it using the `<script>` tag.

Step 9

Our application is ready, but we wrote our code in TypeScript; it has to be compiled into JavaScript. Once this is done, we need the web server to run our application.

Once again, if we refer to `package.json` in `Step 3` in the `devDependencies` section, we specified the `typescript`, `lite-server`, and `concurrently` packages. These are downloaded during the `npm install` execution.

The `typescript` package contains its compiler; `lite-server` package is lightweight node-based web server and concurrently package let us run multiple commands concurrently. Now if we check out `scripts` section, there are three commands:

```
"scripts": {
  "prestart": "npm run build",
  "start": "concurrently \"tsc -w\"  \"lite-server\"",
  "build": "tsc"
}
```

The `build` command is invoking the TypeScript compiler using the `tsc` command. The `start` command is invoking the TypeScript compiler in watch mode, also simultaneously it's running the `lite-sever` command, which will launch web server. The `prestart` command is invoked automatically before the `start` command, and we are simply invoking the `build` command in it.

We can invoke these commands in the scripts section using `npm run <command>`; npm predefined commands can run using `npm <command>`.

Step 10

Finally let's run our first Angular application, run the following command at the root folder of our application:

```
$ npm start
```

The preceding command will compile our TypeScript code and launch the web server. The web server will start our application and launch the default browser and will display the `Say Hello to Angular` message in the browser.
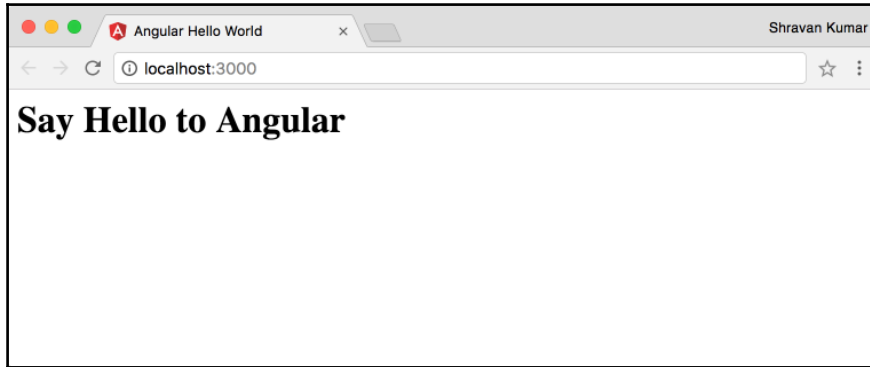
So, what happened? Here are the following steps happened in between once web server launched, we saw the output in the browser.

As soon as a web server loads the `index.html` page, the browser loads the files specified in `scripts` tags. Once browser loads the SystemJS, it read its configuration and loads the `app.js` file (which is compiled from `app.ts` file). In the `app.js` file, `@angular/core`, `@angular/platform-browser`, `@angular/platform-browser-dynamic` modules are loaded; they will load their dependent modules.

The `platformBrowserDynamic()` function bootstraps our `HelloWorldAppModule`, while bootstrapping `HelloWorldAppModule`, it will add the `HelloWorldAppComponent` in the `bootstrap` property to its `entryComponents`.

Once the `HelloWorldAppComponent` is added to `entryComponents`, Angular will compile it and create a `ComponentFactory` instance and store it in the `ComponentFactoryResolver` instance.

Finally, the `<hello-world-app>` tag in `index.html` will render the template of `HelloWorldAppComponent` output.

> ES2015 features will be explained whenever they are used in the code.

Using Angular CLI

In the previous section *Writing your first Angular application*, we learned how to write a hello world application from scratch. To write a simple hello world application, we initially had to create a lot of files with boilerplate code and project configuration. This process is common for both small and large applications.

For large applications, we create a lot of modules, components, services, directives, and pipes with boilerplate code and project configuration. This is a very time-consuming process. Since we want to save time and be productive by focusing on solving business problems instead of spending time on tedious tasks, tooling comes in handy.

The Angular team created a command-line tool know as **Angular CLI**. The Angular CLI helps us in generating Angular projects with required configurations, boilerplate code, and also downloads the required node packages with one simple command. It also provides commands for generating components, directives, pipes, services, classes, guards, interfaces, enums, modules, modules with routing and building, running and testing the applications locally.

Getting started with Angular CLI

The Angular CLI is available as a node package. First, we need to download and install it with the following command:

```
$ npm install -g @angular/cli
```

The preceding command will install Angular CLI, we can then access it anywhere via command line or Terminal. To generate the Angular project using CLI we can use the `ng new project-name` command.

```
$ ng new first-ng-cli-project
```

This command creates a folder named `first-ng-cli-project`, generates Angular project under it with all the required files and downloads all the node packages. To run the application, we need to navigate to the project folder and run the `ng serve` command:

```
$ cd first-ng-cli-project
$ ng serve
```

The `ng serve` command compiles and builds the project, and starts the local web server at `http://localhost:4200` URL. When we navigate to `http://localhost:4200` URL in the browser, we see the following output:
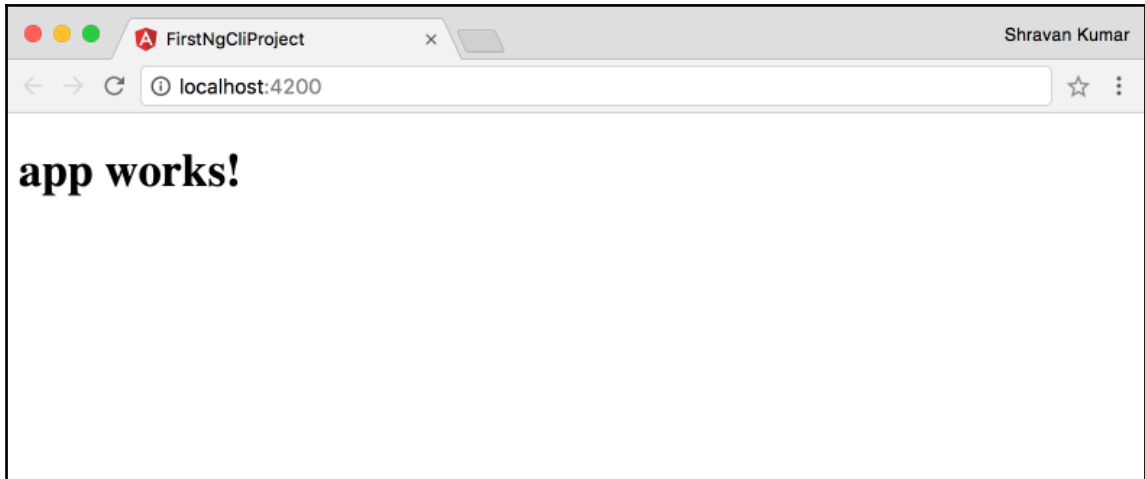
The output is very simple but we generated the entire project, and got it up and running with two simple commands. Here are some more commands to generate different types of files using Angular CLI.

Component	`ng g component my-new-component`
Directive	`ng g directive my-new-directive`
Pipe	`ng g pipe my-new-pipe`
Service	`ng g service my-new-service`
Module	`ng g module my-module`
Module with routing	`ng g module my-module --routing`

To learn more about Angular CLI visit `https://cli.angular.io`.

Summary

We started this chapter with an introduction to Angular, and some of its new features. Then, we discussed few tools that will help us during our application development. Next, we looked at different language choices available to build Angular applications and also learned TypeScript and its importance. We learned how to set up Angular for development and wrote our first Angular application. Finally we looked at Angular CLI and how it accelerates the project development.

In the next chapter, we will learn how to write components, new directives and new templating syntax in Angular.

2

Basics of Components

In this chapter, we will learn how to use new features of the Angular framework, line data binding, event binding, and built-in directives to build components. After going through this chapter, the reader will understand the following concepts:

- How to write components in Angular
- Data binding in Angular
- Templating syntax in Angular
- Built-in directives in Angular

Getting started

Angular has been completely re-written from scratch, so there are a lot of new concepts. In this chapter, we will discuss some of the important features like data binding, new templating syntax, and built-in directives. We are going to use a more practical approach for learning these new features.

Project setup

Here is a sample application with the required configuration for Angular and sample code. Create the directory structure and files as mentioned here:

```
interpolation-syntax

├── index.html
├── package.json
├── src
│   ├── app.component.ts
```

```
|   ├── app.module.ts
|   └── main.ts
├── systemjs-angular-loader.js
├── systemjs.config.js
└── tsconfig.json
```

We can use the same code for `tsconfig.json` and `package.json` files from `Chapter 1`, *Getting Started*, just change the name property to `interpolation-syntax` in the `package.json` file.

In the `Chapter 1`, *Getting Started*, *Hello World* example, we have all our code only in one file. However, once our application starts growing, maintainability becomes a problem, so we are going to split the code into three files:

- `src/main.ts`: It contains all the bootstrapping logic, as follows:

```
import { platformBrowserDynamic } from
                      '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

- `src/app.module.ts` : This will have application module logic:

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

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

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

- `src/app.component.ts`: It contains actual application component logic:

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

@Component({
  selector: 'display-data-app',
  template: '<h1>Data Binding in Angular –
                     Interpolation Syntax</h1>'
})
export class AppComponent {}
```

The code for `index.html` is as follows:

```html
<html>
<head>
  <title>Data Binding in Angular - Interpolation Syntax</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
                                          initial-scale=1">

  <script src="node_modules/core-js/client/shim.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

  <script src="systemjs.config.js"></script>
  <script>
    System.import('src').catch(function (err) {
      console.error(err);
    });
  </script>
</head>
<body>
<display-data-app>Loading...</display-data-app>
</body>
</html>
```

We moved our SystemJS configuration code to a separate file (`systemjs.config.js`) from `index.html`.

The code for `systemjs.config.js` is as follows:

```javascript
(function (global) {
  System.config({
    paths: {
      'ng:': 'node_modules/@angular/'
    },
    map: {
      '@angular/core': 'ng:core/bundles/core.umd.js',
      '@angular/common': 'ng:common/bundles/common.umd.js',
      '@angular/compiler': 'ng:compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'ng:platform-
                        browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'ng:platform-browser-
                 dynamic/bundles/platform-browser-dynamic.umd.js',
      'rxjs': 'node_modules/rxjs'
    },
    packages: {
      src: {
```

```
           main: 'main',
           defaultExtension: 'js',
           meta: {
             './*.js': {
               loader: 'systemjs-angular-loader.js'
             }
           }
        },
        rxjs: {
           defaultExtension: 'js'
        }
      }
    });
  })(this);
```

The `systemjs-angular-loader.js` file contains logic for loading the template and CSS files with relative paths in the component. We can copy this from the provided source code.

We have our application ready, so now run the `npm install` command. Once it is finished, run the `npm start` command. This launches our application in the browser. Up till now, what we have created is no different from the *Hello World* application in `Chapter 1`, *Getting Started*, except for the display message:
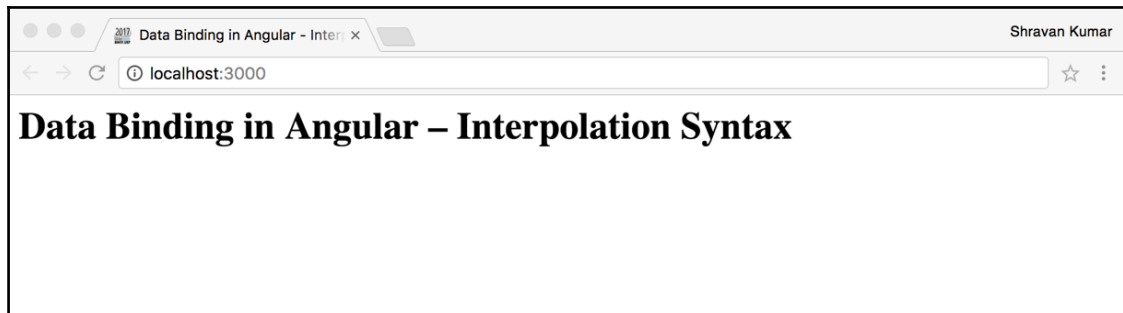
Working with data

In a web application, we need to display data on an HTML page and read the data from input controls on an HTML page. In Angular, everything is a component; the HTML page is represented as a `template`, and it is always associated with a `Component` class. Application data lives on the component's class properties.

Either we push values to the `template` or pull values from the `template`. To do this we need to bind the properties of the `Component` class to the controls on the `template`. This mechanism is known as data binding. Data binding in Angular allows us to use simpler syntax to push or pull data.

Displaying data

In this section, we are going to look at the different syntaxes provided by Angular to display data.

Interpolation syntax

If we remember our `AppComponent` class from the preceding example, we are displaying a message statically in HTML inside the `template`. Let us learn how to show the same message stored in a class property.

Here is the revised code for the `AppComponent` class in the `src/app.component.ts` file:

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

@Component({
  selector: 'display-data-app',
  template: '<h1>{{message}}</h1>'
})
export class AppComponent {
message: string = 'Data Binding in Angular
                   - Interpolation Syntax';
}
```

Once we have the preceding code snippet, the browser will automatically refresh with the latest output, so we do not need to reload the browser manually. This happens because we are running our TypeScript compiler in watch mode, it will automatically compile any modified TypeScript files to JavaScript files.

We are also using a lite-server as our web server, and this will automatically refresh the browser if any of the files under the application change. We can continuously modify the code and view the output without reloading the browser.

In the `Component()` function, we updated the `template` property with an expression `{{message}}` surrounded by an `h1` tag.

The double curly braces are the interpolation syntax in Angular.

For any property on the class that we need to display on the `template`, we can use the property name surrounded by double curly braces. Angular will automatically render the value of the property in the browser.

Let's further extend our example, and bind the message property to a text box. Here is the revised `template`:

```
template: `
  <h1>{{message}}</h1>
  <input type="text" value="{{message}}"/>
`
```

Notice that the preceding `template` is a multiline string, and it is surrounded by `` ` `` (backtick) symbols instead of single or double quotes.

> `` `` `` (backtick) symbols are the new multiline string syntax in ECMAScript 2015.
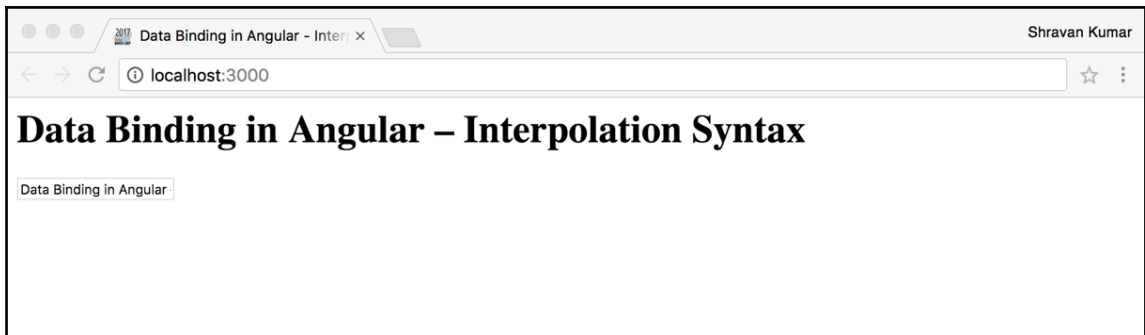
Now the text box is also displaying the same value in the message property. Let's change the value in the text box by typing something, then hit the *Tab* button. We do not see any changes happening in the browser.

Whenever we modify the value of any control on the `template`, which is bound to a property of a `Component` class, it should update the property value. Any other controls bound to the same property should also display the updated value on the `template`.

In the browser, the `<h1>` tag should also show the same text whatever we type in the text box, but this will not happen.

Interpolation syntax is one-way data binding, and data flows from the data source (`Component` class) to view (`template`).

Only the value of the property is updated on the `template`, it will not happen vice-versa, that is, changes made to controls on the `template` will not update the property value.

Property binding

Create an example `property-binding` from the previous example. Just change the name property to `property-binding` in the `package.json`. Then run the `npm install` command followed by the `npm start` command. Create the directory structure and files as mentioned here:

```
property-binding
├── index.html
├── package.json
├── src
│   ├── app.component.ts
│   ├── app.module.ts
│   └── main.ts
├── systemjs-angular-loader.js
├── systemjs.config.js
└── tsconfig.json
```

Property binding is another form of data binding syntax in Angular.

```
<element-name [element-property-name] = "component-property-name">
```

Property binding syntax:

- `element-name`: This can be any HTML tag, or custom tag
- `element-property-name`: Specifies the property of the corresponding DOM element for the HTML tag or custom tag property name surrounded by square brackets
- `component-property-name`: Specifies the property of the component class or expression

Here is an example of property binding in Angular: `<img [src]="headerImage" />`

In the preceding code snippet, we are binding the `headerImage` property of the `Component` class to the `src` attribute of the `<img />` tag.

An important point to remember is that unlike AngularJS 1, Angular does not bind values to attributes of HTML elements. Instead, it will bind to properties of corresponding **DOM** (**Document Object Model**) elements.

In the preceding code snippet for the `<img />` HTML tag, `HTMLImageElement` is the corresponding interface in DOM. For most of the HTML element's attributes, there will be one to one mapping with its corresponding DOM interface properties, but there are exceptions. Property binding works with only properties, not attributes.

Let us update our `template` in the `property-binding` example to use property binding:

```
template: `
  <h1 [textContent]="message"></h1>
  <input type="text" [value]="message"/>
```

Instead of using interpolation syntax, we are wrapping the `textContent` property of the `<h1>` tag and value property input tag in square braces, and on the right side of this expression, we are assigning the `Component` class properties. The output will be the same as when we are using interpolation syntax.

> Property binding syntax is also one-way data binding, data flows from data source (`Component` class) to view (`template`).

For the property binding syntax and interpolation syntax, we can use them interchangeably; there are no differences other than syntax. We can use whichever syntax is more idiomatic for a given situation.

Instead of using square brackets notation for property binding, we can also use its canonical form property name prefixed with `bind-`:

```
<h1 bind-textContent ="message"></h1>
```

Attribute binding

Angular always uses properties to bind the data. But if there is no corresponding property for the attribute of an element, Angular will bind data to attributes. Attribute binding syntax starts with the keyword `attr` followed by the name of the attribute and then assigns it to the property of the `Component` class or an expression:

```
<td [attr.colspan]="colSpanValue"></td>
```

Event binding

Using event binding syntax, we can bind built-in HTML element events, such as `click`, `change`, `blur`, and so on, to `Component` class methods. We can also bind custom events on components or directives, which we will discuss in the chapters that follow.

Event binding syntax uses parenthesis symbols `()`. We need to surround the event property name with parenthesis symbols `()` on the left side of the expression, on the right side we will specify one of the `Component` methods which will be invoked when the event is triggered.

Create another example `event-binding` from the previous example. Change the name property to `event-binding` in the `package.json` file. Then run the `npm install` command followed by the `npm start` command.

The code for the revised `AppComponent` class is as follows:

```
export class AppComponent {
  public message: string = 'Angular - Event Binding';

  showMessage() {
    alert("You pressed a key on keyboard!");
  }
}
```

We have added a method named `showMessage()` to the `AppComponent` class, this method will be invoked whenever we type a key in the text box.

The code for the revised `template` on `AppComponent` is as follows:

```
template: `
  <h1>{{message}}</h1>
  <input type="text" [value]="message"
                      (keypress)="showMessage()"/>
`
```

We have added a `keypress` event surrounded by parenthesis symbols on the text box to bind with the `showMessage()` method in the `AppComponent` class. Let's update our example to be a little bit more realistic, instead of displaying the same `alert()` dialog every time we display the keys we are typing:

The code for `src/app.component.ts` is as follows:

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

@Component({
  selector: 'event-binding-app',
  template: `
    <p>{{message}}</p>
    <input type="text" (keypress)="showMessage($event)"/>
  `
})
export class AppComponent {
  public message: string = 'Angular - Event Binding';

  showMessage(onKeyPressEvent) {
    this.message = onKeyPressEvent.target.value;
  }
}
```

Here are the important things to notice in our code:

- To the `showMessage` method, we are passing a special Angular `$event` object
- `$event` keyword represents the current DOM event object
- On the `AppComponent` class `showMessage` method, we are accepting `$event` passed from `template` into the `onKeyPressEvent` method parameter
- Every DOM event object has a `target` property, which represents the DOM element on which the current event is raised
- We are using the `onKeyPressEvent.target` object, which represents the text box
- We are using the `onKeyPressEvent.target.value` property to access to the text box value
- We are assigning the value of the text box to the `message` property

As a result of the preceding code, whatever input we enter in the text box will appear in the `<p>` tag output because it is also bound to the `message` property, we are updating the `message` property value whenever we type something into the text box.

> Event binding syntax is also one-way data binding, but the data flows from view (`template`) to the `Component` class.

Instead of using `()` symbols notation for event binding, we can also use its canonical form event name prefixed with `on-`:

```
<input type="text" on-keypress="showMessage($event)"/>
```

We will discuss event binding again in future chapters.

Two-way data binding

We require the data to flow in both directions, from `Component` to `template` and vice-versa. The most classic example is forms. We need to display the properties' values on views, and when the user updates the values on views, we need them to be updated back to properties.

Two-way data binding syntax is the combination of both property binding and event binding along with the `ngModel` built-in directive. We need to surround the `ngModel` directive with both parenthesis and square brackets `[(ngModel)]`:

```
[(ngModel)] = "component-property"
```

Create another example `two-way-binding` from the previous example. Change the name property to `two-way-binding` in the `package.json` file.

We need to do a few more things for two-way data binding to work, and we require the `@angular/forms` package; the `ngModel` directive is available in it. Run the following command at the root of the project to install the `@angular/forms` package:

```
$ npm install @angular/forms --save
```

The preceding command will download the `@angular/forms` package and also add the entry in the dependencies section in the `package.json` file. Then add the following line to the `systemjs.config.js` map object, so that SystemJS will load the `forms` module:

```
'@angular/forms': ng:forms/bundles/forms.umd.js'
```

Include the `FormsModule` to the `imports` arrays, our `AppModule` is dependent on the forms module:

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

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

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

Now we have everything ready for us to use `ngModel`, let us run the `npm install` command followed by the `npm start` command.

The codes for `src/app.component.ts` are as follows:

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

@Component({
  selector: 'two-way-binding-app',
  template: `
    <p>{{message}}</p>
    <input type="text" [(ngModel)]="message" />
  `
})
export class AppComponent {
  public message: string = 'Angular - Two Way Binding';
}
```

In the preceding code snippet, `message` property is assigned to `[(ngModel)]`, if we change the text in the text box, the `message` property will be updated automatically without any extra lines of code and vice versa. This also updates the text inside the `<p>` tag.

Built-in directives

The AngularJS 1 framework has a lot of built-in directives. Angular comes with very few directives, the remaining directives in AngularJS 1 are replaced with new concepts of Angular which we discussed in previous sections. In this section, we will discuss the built-in directives available in Angular.

Structural directives

The structural directives allow us to change the DOM structure in a view by adding or removing elements. In this section, we will explore built-in structural directives, `ngIf`, `ngFor`, and `ngSwitch`.

ngIf

The `ngIf` directive is used for adding or removing elements from DOM dynamically:

```
<element *ngIf="condition"> content </element>
```

If the condition is true, Angular will add content to DOM, if the condition is false it will physically remove that content from DOM:

```
<div *ngIf="isReady">
   <h1>Structural Directives</h1>
   <p>They lets us modify DOM structure</p>
</div>
```

In the preceding code snippet, when `isReady` value is true, the content inside the `<div>` tag will be rendered on the page, whenever it is false, both tags inside the `<div>` tag will be completely removed from DOM. The asterisk (*) symbol before `ngIf` is a must.

ngFor

The `ngFor` is a repeater directive, it's used for displaying a list of items. We use `ngFor` mostly with arrays in JavaScript, but it will work with any iterable object in JavaScript. The `ngFor` directive is similar to the `for...in` statement in JavaScript.

Here is a quick example:

```
public frameworks: string[]  = ['Angular', 'React', 'Ember'];
```

The framework is an array of frontend framework names. Here is how we can display all of them using `ngFor`:

```
<ul>
  <li *ngFor="let framework of frameworks">
    {{framework}}
  </li>
</ul>
```

The preceding code snippet will display the list of framework names in an unordered list.

Understanding ngFor syntax

```
<li *ngFor="let framework of frameworks">
    {{framework}}
</li>
```

The preceding code uses `ngFor` to display the list of framework names. Let us understand each part of the `ngFor` syntax:

```
*ngFor="let framework of frameworks"
```

There are multiple segments in the `ngFor` syntax, which are `*ngFor`, `let framework`, and `frameworks`. We will now see them in detail:

- `frameworks`: This is a array and data source for the `ngFor` directive on which it will iterate.
- `let framework`: `let` is a keyword used for declaring the `template` input variable. The `template` input variable represents a single item in the list during iteration. We can use a `framework` variable inside an `ngFor template` to refer to the current item of iteration.
- `*ngFor`: `ngFor` represents the directive itself, the asterisk (`*`) symbol before `ngFor` is a must.

ngSwitch

The `ngSwitch` directive behaves similarly to a switch case statement in JavaScript. `ngSwitch` will have multiple templates, depending on the value passed, it will render one template. This directive is similar to the `switch()` statement in JavaScript:

```
<div [ngSwitch]="selectedCar">
  <template [ngSwitchCase]="'Bugatti'">I am Bugatti</template>
  <template [ngSwitchCase]="'Mustang'">I am Mustang</template>
  <template [ngSwitchCase]="'Ferrari'">I am Ferrari</template>
  <template ngSwitchDefault>I am somebody else</template>
</div>
```

We are using property binding syntax with `[ngSwitch]`. In the preceding code snippet, when the `selectedCar` property value matches the `[ngSwitchCase]` value, Angular will render that template content physically, the remaining templates won't be on the screen. If none of the `[ngSwitchCase]` values match, Angular will render the `ngSwitchDefault` template.

We are not using the asterisk (*) symbol for NgSwitch because we are directly using the HTML 5 `template` tag. `ngIf` and `ngFor` also use an HTML 5 `template` tag internally for rendering content, instead of explicitly writing the `template` tag every time. Unlike `ngSwitch`, we use the asterisk (*) symbol as a shortcut or syntactic sugar. Angular will internally replace the asterisk (*) symbol with the HTML 5 `template` tag.

Attribute directives

The attribute directives allow us to change the appearance or behavior of an element. In this section, we will explore the built-in attribute directives, `ngStyle`, and `ngClass`.

ngStyle

The `ngStyle` directive is used when we need to apply multiple inline styles dynamically to an element.

In the `template`:

```
<p [ngStyle]="getInlineStyles(framework)">{{framework}}</p>
```

In the `Component` class:

```
getInlineStyles(framework) {
  let styles = {
    'color': framework.length > 3 ? 'red' : 'green',
    'text-decoration': framework.length > 3 ? 'underline' : 'none'
  };
  return styles;
}
```

Instead of writing lengthy statements at `[ngStyle]` in the template, we are calling a method inside the `Component` class, which returns multiple inline styles.

If we need to apply a single inline style dynamically, we can use style binding using the following syntax, `[style.style-property]` instead of the `ngStyle` directive:

```
<p [style.color]="framework.length > 3 ? 'red': 'green'" >
  {{framework}}
</p>
```

> The `let` is a new keyword part of ES2015, which will allow us to declare a block level scope local variable. Before ES2015 there is no block level scope in JavaScript.

ngClass

The `ngClass` directive is used when we need to apply multiple classes dynamically.

The code for `Styles` is as follows:

```
.red {
  color: red;
  text-decoration: underline;
}

.bolder {
  font-weight: bold;
}
```

In the `Component` class:

```
geClasses(framework) {
  let classes = {
    red: framework.length > 3,
    bolder: framework.length > 4
  };
  return classes;
}
```

In the `template`:

```
<p [ngClass]="geClasses(framework)">{{framework}}</p>
```

Whichever classes are true, they will be applied to the template. If we need to apply a single class dynamically, we can use class binding using the following syntax; `[class.class-name]` instead of the `ngClass` directive:

```
<p [class.red]="isThisRed">{{framework}}</p>
```

Building the master-detail component

We have learned a lot of new things in Angular, such as data binding, event binding, structural directives, and attribute directives in this chapter. Let's put them into action by building a master-detail page application.

Let us begin by creating another example from the previous section and naming it `master-details`. Change the name property to `master-details` in the `package.json` file. We do not need the forms package in this example. Finally, run the `npm install` command followed by the npm start command.

The code for `index.html` is as follows:

```html
<html>
<head>
  <title>Books List</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
                                  initial-scale=1">

  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/
          4.0.0-alpha.6/css/bootstrap.min.css" rel="stylesheet">
  <link href="https://fonts.googleapis.com/css?family=Roboto:
                                  400,500" rel="stylesheet">
  <link rel="stylesheet" href="styles.css"/>

  <script src="node_modules/core-js/client/shim.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

  <script src="systemjs.config.js"></script>
  <script>
    System.import('src').catch(function (err) {
      console.error(err);
    });
  </script>
</head>
<body>
  <books-list>Loading...</books-list>
</body>
</html>
```

We added three additional items to `index.html`:

- **Bootstrap**: CSS frameworks from CDN (we can use any CSS)
- **Roboto font**: We can use any font we like
- **Custom style sheet**: Where we write our application styles

> **Code for styles.css**
> The code for stylesheet (`styles.css`) is very long. The reader can add it from the provided source code.

The code for `src/app.module.ts` is as follows:

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

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

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

The code for `src/app.component.ts` is as follows:

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

@Component({
  selector: 'books-list',
  templateUrl: ./app.component.html'
})
export class AppComponent {
}
```

There is one new thing in our `AppComponent`, that is, `templateUrl`. Instead of using an inline template, we are using the template stored in an HTML file.

The code for `src/app.component.html` is as follows:

```html
<h3 class="title">Books List</h3>
<div class="border-divider"></div>
<div class="row">
    <div class="col-xs-3">Left</div>
```

```
    <div class="col-xs-9">Right</div>
  </div>
```

We created a basic layout structure using bootstrap in the preceding HTML file. Let us write some code in this example. We are going to display basic book information on the left-hand side of the page. The user will be able to click on any of the book items and pull information about that book, and the information gets displayed on the right-hand side of the page.

The next step is to add the `Book` class to our example to define the structure of it.

The code for `src/book.ts` is as follows:

```
export class Book {
  isbn: number;
  title: string;
  authors: string;
  published: string;
  description: string;
  coverImage: string;
}
```

Also, let us add some sample data. Normally, this data comes from REST services in real-world applications, we will learn how to consume and use them in future chapters. But for this example, we are going to use dummy data inside a class.

The code for `src/mock-books.ts` is as follows:

```
import { Book } from './book';

export const BOOKS: Book[] = [
  {
    isbn: 9781786462084,
    title: 'Laravel 5.x Cookbook',
    authors: 'Alfred Nutile',
    published: 'September 2016',
    description: 'A recipe-based book to help you efficiently
    create amazing PHP-based applications with Laravel 5.x',
    coverImage: 'https://d255esdrn735hr.cloudfront.net/sites/
    default/files/imagecache/ppv4_main_book_cover/
    B05517_MockupCover_Cookbook_0.jpg'
  },
  {
    isbn: 9781784396527,
    title: 'Sitecore Cookbook for Developers',
    authors: 'Yogesh Patel',
    published: 'April 2016',
```

```
      description: 'Over 70 incredibly effective and practical
      recipes to get you up and running with Sitecore development',
      coverImage: 'https://d255esdrn735hr.cloudfront.net/sites/'
      default/files/imagecache/ppv4_main_book_cover/6527cov_.jpg'
    },
    {
      isbn: 9781783286935,
      title: 'Sass and Compass Designers Cookbook',
      authors: 'Bass Jobsen',
      published: 'April 2016',
      description: 'Over 120 practical and easy-to-understand
      recipes that explain how to use Sass and Compass to write
      efficient, maintainable, and reusable CSS code for your web
      development projects',
      coverImage: 'https://d1ldz4te4covpm.cloudfront.net/sites/
      default/files/imagecache/ppv4_main_book_cover/I6935.jpg'
    }
];
```

The code for `src/app.component.ts` is as follows:

```
import { Component } from '@angular/core';
import { Book } from './book';
import { BOOKS } from './mock-books';
n
@Component({
  selector: 'books-list',
  templateUrl: 'src/app.component.html'
})
export class AppComponent {
  booksList: Book[] = BOOKS;
}
```

We are storing all the `mock-books` data inside the `books-list` property of the `AppComponent` class. As we know, any public property of the `Component` class is accessed inside the template, so we will use `*ngFor` to display this list inside the template (`app.component.html`):

```
<ul class="list">
<li class="list-item" *ngFor="let book of booksList">
   <div class="cover-image-container">
      <span class="cover-image">
         <img [src]="book.coverImage" alt="cover image"/>
      </span>
   </div>
   <div class="clear">
      <h3 class="book-title">{{book.title}}</h3>
      <h4 class="book-author">{{book.authors}}</h4>
```

```
        </div>
    </li>
</ul>
```

The preceding code snippet needs to be added in the place of text which stays left. Here we are using `*ngFor` for iterating over the `booksList`. Using the `template` input variable book interpolation syntax, we are displaying the image, book title, and authors of the book. This is how our example looks in a browser at this stage:
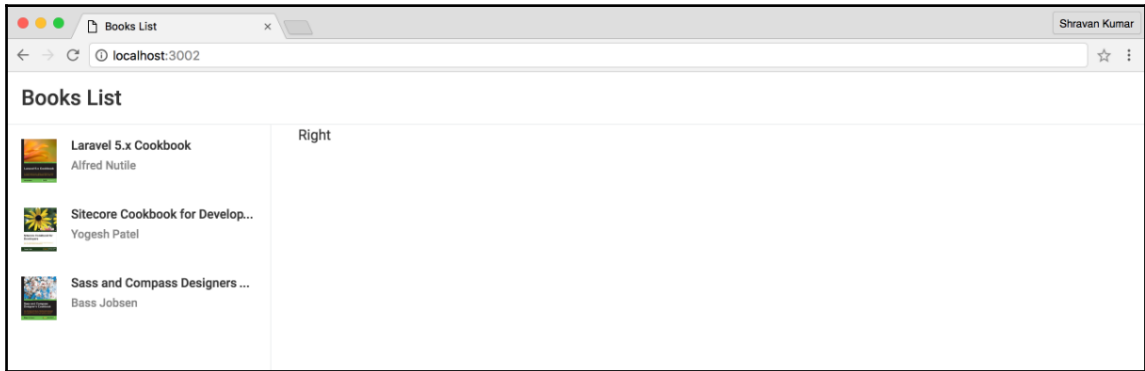
Now, whenever we click on any one of the list items, on the right side should be displayed the full details of the book. We need to add a `click` event to each list item, associate it with a method in `Component` class, and set the `selectedBook` details in the property using that method so that we can use the `selectedBook` property to display the full details of the book on the right side of the template.

Now add the `click` event to each list item:

```
<li class="list-item" *ngFor="let book of booksList"
                    (click)="getBookDetails(book.isbn)">
```

Add the `getBookDetails()` method and property to store the selected book. To the `getBookDetails` method, we are passing a book's ISBN, using which we can filter the details of the book from `booksList`:

```
export class AppComponent {
  booksList: Book[] = BOOKS;
  selectedBook: Book;

  getBookDetails (isbn: number) {
    var selectedBook = this.booksList
.filter(book => book.isbn === isbn);
    this.selectedBook = selectedBook[0];
```

```
    }
  }
```

We are storing our selected book information in the `selectedBook` property of the `Component` class, and we can use the same property in the template to display the details:

```
<div class="col-xs-9">
  <div class="row selected-book">
    <div class="col-xs-2">
      <img [src]="selectedBook.coverImage" alt="cover image"/>
    </div>
    <div class="col-xs-8">
      <h3 class="title">{{selectedBook.title}}</h3>
      <p>{{selectedBook.authors}}</p>
      <p>{{selectedBook.published}}</p>
      <p>{{selectedBook.description}}</p>
    </div>
  </div>
</div>
```

By now, our application is almost complete. If we go and click on any of the list items, on the right side it will display the full details of the book:
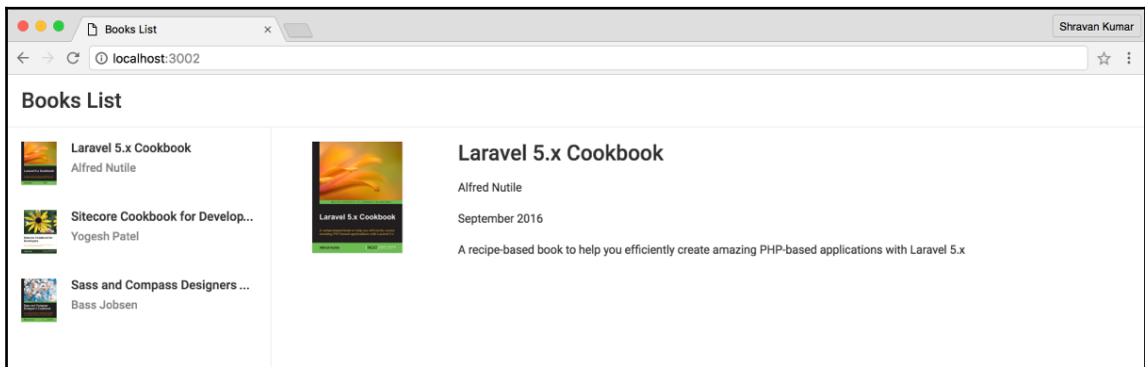
We are all good, except there is one problem: whenever the application is loaded, we do not have a selected book. If we go to developer tools on the **Console** tab in the browser, we will see a bunch of errors; this is happening because we are trying to display the selected book's information whose value is undefined. We should render the template only when there is a value in the `selectedBook` property, which we can simply check by adding a `*ngIf` directive to check the value of the `selectedBook` property:

```
<div *ngIf="selectedBook" class="row selected-book">
```

Now, we don't see any errors because the `*ngIf` directive checks for the value of the `selectedBook` property. If it is not initialized with book details, it won't even render the content inside of it. The template code will not try to access the value of the `selectedBook` property because the code won't even exist. With this, we will end this chapter here.

Summary

We started this chapter by covering an introduction to components. Next, we discussed how to write components using new features in Angular, such as data binding (one-way, two-way), event binding, and new templating syntaxes using many examples. Then we discussed different kinds of built-in directives in Angular. Finally, we completed this chapter by building a master-detail application using all those features we learned throughout this chapter.

By the end of this chapter, the reader should have a good understanding of Angular's new concepts and should be able to write components. In the next chapter, we will discuss components, services, and dependency injection in more depth.

3

Components, Services, and Dependency Injection

In this chapter, we will learn how to implement some real-world application scenarios. We are going to look at how to implement multiple components in the master-detail page, how the components communicate with each other, how to share the data between these components using services, and how dependency injection works. After going through this chapter, the reader will understand the following concepts:

- Creating and using multiple components
- Communication between the components
- Using services to share the data
- How dependency injection work

Introduction

In the previous chapter, we learned how to build a book list application, which is a `master-detail` page. It contains just one component and becomes complex; these components need to communicate with each other.

Let us begin by creating another application from the last example in the previous chapter, name it `multiple-components`, also change the name property to `multiple-components` in the `package.json` file.

Now run the `npm install` command; this will download all the required packages. Once it has finished, run the `npm start` command. We will see the following output in the browser:
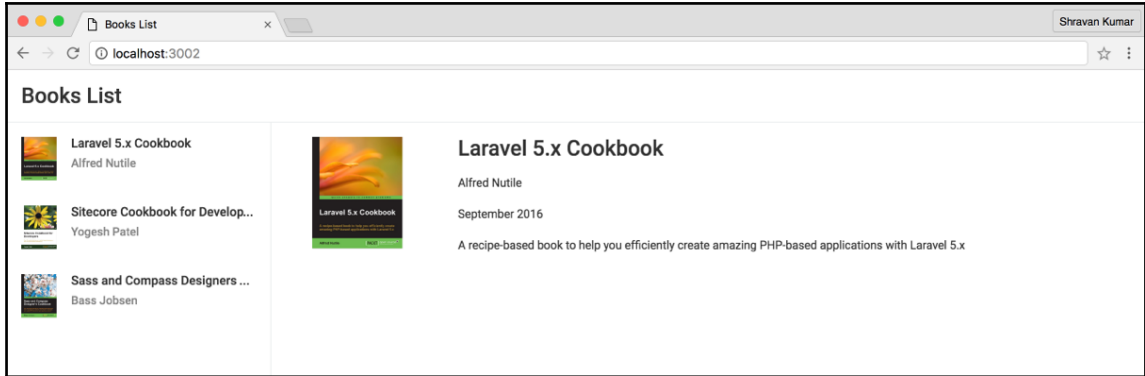
Our application has a single component named `AppComponent`, and its template code is in the `app.component.html` file. A single component is doing too many things in our application. In the future, we might need to display book information in other components. If we have to write the same piece of code again by duplicating, we will violate the **DRY** (**Don't Repeat Yourself**) principle. The component should be an atomic, a reusable UI building block, and it should be as small as possible, and reusable.

Working with multiple components

The real-world applications will be complex, and they will have multiple components. We are going to rewrite our application to use multiple components and understand how these components communicate with each other.

If we look at our template code in the `app.component.html` file, we have some pieces of code at the bottom, inside `<div class="col-xs-9"></div>` tags, which displays the selected book information. We need to refactor this code into a separate component (`BookDetailsComponent`) so that it is reusable in multiple places as needed.

Let us create a folder, `book-details` under the `src` folder, then create a file, `book-details.component.ts` under the `book-details` folder. We are going write all the code inside `<div class="col-xs-9"></div>` tags in the template of `BookDetailsComponent`.

The code for `src/book-details/book-details.component.ts` is:

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

@Component({
  selector: 'book-details',
  templateUrl: './book-details.component.html'
})
export class BookDetailsComponent {
  book: Book;
}
```

The code for `src/book-details/book-details.component.html` is as follows:

```
<div *ngIf="book">
  <div class="row selected-book">
    <div class="col-xs-2">
      <img [src]="book.coverImage" alt="cover image"/>
    </div>
    <div class="col-xs-8">
      <h3 class="title">{{book.title}}</h3>
      <p>{{book.authors}}</p>
      <p>{{book.published}}</p>
      <p>{{book.description}}</p>
    </div>
  </div>
  <div class="row">
    <div class="col-xs-10">
      <button class="btn btn-danger float-xs-right mt-1">
        Delete
      </button>
    </div>
  </div>
</div>
```

We have our new `book-details` component. There are a couple of things to notice; we are using the `book` as a property name in the `Component` class instead of `selectedBook`, the template also uses the same property name. We also added a `Delete` button to the template code which does nothing for now.

Our refactored component can display the given book information. To do that we can use the `book-details` selector declaratively anywhere:

```
<book-details></book-details>
```

To use this `BookDetailsComponent`, first we need to add it to the `declarations` array in our app module:

```
import { BookDetailsComponent } from
                    './book-details/book-details.component';



declarations: [AppComponent, BookDetailsComponent]
```

It is time to use the `BookDetailsComponent` inside the template of `AppComponent`. Here is the `app.component.html` code using `BookDetailsComponent` declaratively, using its selector, `<book-details>` inside `<div class="col-xs-9"></div>` tags.

The code for `src/app.component.html` is as follows:

```
<div class="col-xs-9">
  <book-details></book-details>
</div>
```

The `<book-details></book-details>` tag is supposed to display full book information. To do that, the `Component` class needs a `book` object. However, the book information is in the `selectedBook` property of the `AppComponent` class.

We need to bind the `selectedBook` property of the `AppComponent` class to the `book` property of the `BookDetailsComponent` class using property binding on the `<book-details>` tag.

Input properties

The properties on the `Component` class are not directly accessible for property binding on its `selector`; we need to declare them as input properties for property binding to work.

We can declare a property as an input property using the `@Input()` decorator or by adding the property to the `inputs: []` array property on the `@Component()` decorator; we can use either of these ways.

Let's declare our `book` property of the `BookDetailsComponent` class as an input property using the `@Input()` decorator. It is available in the `@angular/core` package:

```
@Input() book: Book;
```

The code for `src/book-details/book-details.component.ts` is as follows:

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

@Component({
selector: 'book-details',
  templateUrl: './book-details.component.html',
})
export class BookDetailsComponent {
  @Input() book: Book;
}
```

Now the `book` property of the `BookDetailsComponent` class is available for property binding.

The code for `src/app.component.html` is as follows:

```
<div class="col-xs-9">
    <book-details [book]="selectedBook"></book-details>
</div>
```

After adding the preceding code snippet, if we head back to a browser, we can view the full output. Click on the **Books List** on the left-hand side and we get the full book information on the right-hand side. It works as expected.

Now, we have successfully created two components, and we are passing data from parent component to child component. This is how components communicate with each other. The parent component interacts with the child component by binding the `selectedBook` property on the parent component to the `book` property on a child component.

We created a reusable `BookDetailsComponent`, which is a presentation component that can be used anywhere, by passing the book information to the `book` property using property binding.

In this section, we learned how the parent component communicates with a child component. In the next section, we will learn how a child component can communicate back to the parent component.

Aliasing input properties

If we do not want to use the original property name for input property binding, we can use aliasing. The `@Input()` decorator accepts an optional alias name for the property:

```
@Input('myBook') book: Book;
```

We aliased the `book` input property name with the name, `myBook`; now we can use `myBook` as a property name on the selector instead of the property name, `book`:

```
<book-details [mybook]="selectedBook"></book-details>
```

Output properties

Let us add the following `deleteBook()` method to the `AppComponent` class. This method deletes book information from `booksList` with a given ISBN number:

```
deleteBook (isbn: number) {
  this.selectedBook = null;
  this.booksList = this.booksList
                      .filter(book=>book.isbn !== isbn);
}
```

If we revisit our `BookDetailsComponent`, we have a **Delete** button in the template. Click on the `Delete` button and nothing happens because we did not wire any events to that button. Add an empty `deleteBook()` method without any implementation to the `BookDetailsComponent` class:

```
export class BookDetailsComponent {
  @Input() book: Book;

  deleteBook () {
  }
}
```

Add a `(click)` event to the `Delete` button in the template and wire the `deleteBook()` method:

```
<button class="btn btn-danger float-xs-right mt-1"
        (click)="deleteBook()">Delete
</button>
```

Now, if we click on the **Delete** button we are still invoking the `deleteBook()` method, it does nothing because we did not need to implement the delete functionality. In fact, `BookDetailsComponent` does not know how to remove a book; it does not have any knowledge of `booksList` from where it is supposed to remove the book information. It is just a presentation component.

The data source `booksList` and logic to delete the book information from the data source are in our parent, `AppComponent`. Whenever we click on the `Delete` button in `BookDetailsComponent` (child component), we need to communicate back to `AppComponent` (parent component) to invoke the `deleteBook()` method on it.

We need to define an event on the `BookDetailsComponent` (child component) which can trigger the `deleteBook()` method on the `AppComponent` (parent component) class. In the `BookDetailsComponent` (child component), create a custom event called `onDelete`.

The `onDelete` property is an output property declared using the `@Output()` decorator to make the event binding work on the `selector`. Custom events can be created using the `EventEmitter` class. The `@Output()` decorator and `EventEmitter` class are available in the `@angular/core` package.

The code for `src/book-details/book-details.component.ts` is as follows:

```
import { Component, Input, Output, EventEmitter }
                                   from '@angular/core';
import { Book } from '../book';

@Component({
  selector: 'book-details',
  templateUrl: './book-details.component.html'
})
export class BookDetailsComponent {

  @Input() book: Book;

  @Output() onDelete = new EventEmitter<number>();

  deleteBook() {
  }
}
```

Whenever the **Delete** button is clicked, we need to trigger an `onDelete` event, the `EventEmitter` class provides an emit method to trigger the events. We are invoking the `deleteBook()` method when we click on the **Delete** button. Let us trigger an `onDelete` event in the `deleteBook()` method using the `emit()` method:

```
deleteBook () {
  this.onDelete.emit(this.book.isbn);
}
```

To `emit()`, we are passing the current book ISBN number. Now we can use the `onDelete()` event to trigger the parent `AppComponent` class' `deleteBook()` method and pass the ISBN number. Add an `onDelete` event on the `<book-details>` selector in the `app.component.html` template file:

```
<div class="col-xs-9">
  <book-details [book]="selectedBook"
                (onDelete)="deleteBook($event)">
  </book-details>
</div>
```

Now the **Delete** button works. Let's understand what we did step by step:

- An `onDelete` event is declared as output property using the `@Output()` decorator
- The `onDelete` property is initialized as an instance of the `EventEmitter` class
- `EventEmitter` objects are used for creating and triggering custom events
- The `onDelete` property is an event, so we are using event binding syntax to bind to `deleteBook()` method in the parent component
- In the `deleteBook()` method of the child component, we are using the `EventEmitter` object and `emit()` method on the `onDelete` property, we are passing the current book's ISBN number as a parameter
- When an `onDelete` event is triggered from the child, it will invoke the `deleteBook()` method on `AppComponent`
- We are passing the ISBN number using the `$event` object to the `deleteBook()` method on `AppComponent`

> While creating the `EventEmitter` object, we are using a number type because we are passing a number via our event, we can use any type in the component.

Aliasing output properties

If we do not want to use the original event name for event binding, we can use aliasing. The `@Ouput()` decorator accepts an optional alias name for the property:

```
@Output('deleteMyBook') onDelete = new EventEmitter<number>();
```

We aliased the `onDelete` output property name with the name, `deleteMyBook`; now we can use `deleteMyBook` as an event name on the selector instead of `onDelete`:

```
<book-details (deleteMyBook)="deleteBook($event)">
</book-details>
```
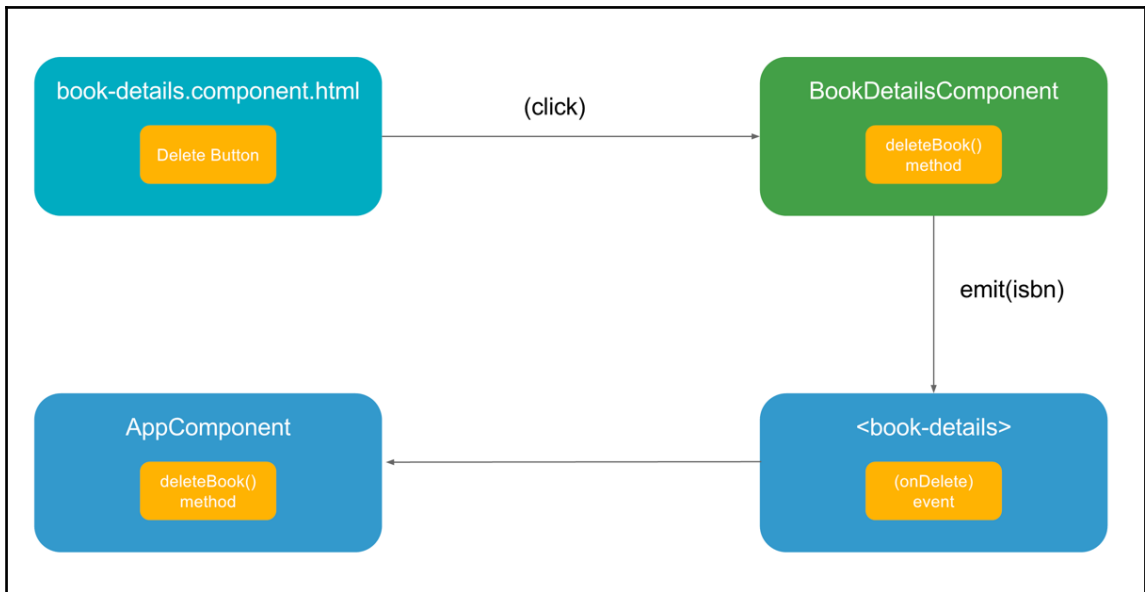

Now, we have successfully implemented communication between parent component and child component in both directions:

The arrows in the diagram represent data flow direction.

Sharing data using services

In our previous example, we have our books sample data in the `mock-books.ts` file, we are accessing data in it directly in `AppComponent`. In real-world applications, we will access the data from external data sources via rest services. We need to access the same data and its operations in multiple components. We need a single, reusable data access point, and this can be implemented as a service in Angular.

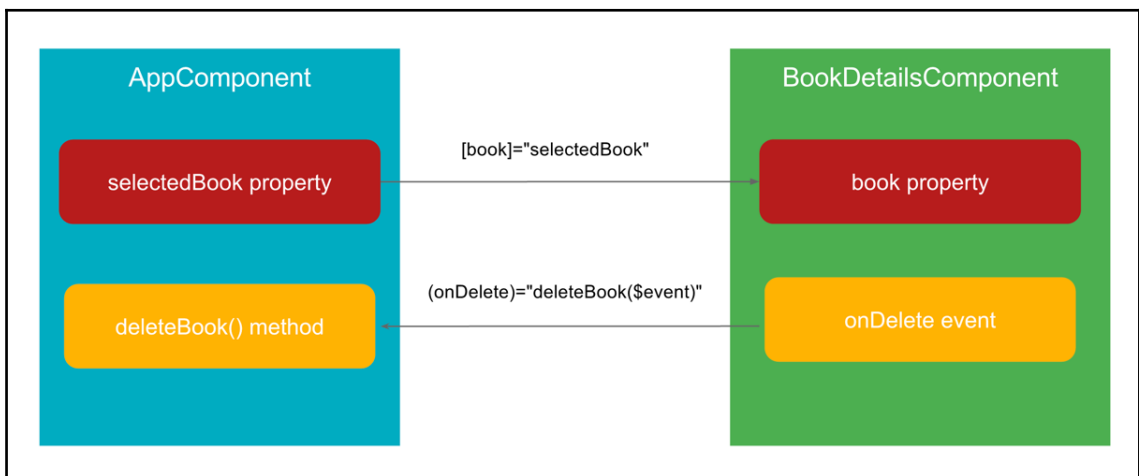A service in Angular written using TypeScript is simply a class, which acts as a reusable data access point. Let's refactor our data access logic into a service to fetch the data, filter the data, and delete data; we were doing all these operations earlier in the component. Once we move them to a service, they can be accessed anywhere within the components.

First, begin with creating another example from the last example, name it `services`, change the name property to `services` in the `package.json` file to reflect the appropriate example name.

The code for `src/book-store.service.ts` is as follows:

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

import { Book } from './book';
import { BOOKS } from './mock-books';

@Injectable()
export class BookStoreService {

  booksList: Book[] = BOOKS;

  getBooks () {
    return this.booksList;
  }

  getBook (isbn: number) {
    var selectedBook = this.booksList
                          .filter(book => book.isbn === isbn);
    return selectedBook[0];
  }

  deleteBook (isbn: number) {
    this.booksList = this.booksList
                          .filter(book => book.isbn !== isbn);
```

```
      return this.booksList;
    }
  }
```

`BookStoreService` contains logic for fetching the books list, filtering a single book, and deleting a book. There is nothing special about this class, it is simply a TypeScript class with methods operating on the `booksList` property, which is our data source.

In real-world applications, these methods might communicate with external rest services using mechanisms like XHR and JSONP. The underlying logic can be changed anytime without affecting components which are consuming a service as long as we do not change the method signatures.

There is one noticeable thing, the `@Injectable()` decorator. The `@Injectable()` decorator is used by TypeScript to emit metadata about our service, metadata that Angular needs to inject other dependencies into this service. Our `BookStoreService` does not have any dependencies at the moment, but we are adding the decorator as it is a best practice for consistency in our code and might be useful in future.

Now we need to refactor our `AppComponent` to use `BookStoreService` methods. First, we need to create a `BookStoreService` object. Unlike any other class, we can create an object to `BookStoreService` using a new operator and its constructor inside the `AppComponent` class:

```
var bookStoreService = new BookStoreService();
```

The preceding code snippet creates the `BookStoreService` object, but also creates dependency, or tight coupling, between `AppComponent` and `BookStoreService`. If the `bookStoreService` constructor definition changes, we need to update the `AppComponent` and all other components wherever we create an object for this service. `bookStoreService` might be dependent on other services; we also need to manage those dependencies.

We do change the definitions in real-world applications, managing all these dependencies between services, directives, and components is difficult, and our code quickly becomes unmanageable and unit testing becomes tough. This is where dependency injection comes into play. Instead of creating objects for dependencies, they can be passed to dependent object constructor:

```
export class AppComponent {
  constructor (private bookStoreService: BookStoreService) {
  }
}
```

Someone needs to create the object for `BookStoreService` and pass it an `AppComponent` constructor. Angular comes with its own dependency injection system.

Instead of creating the `BookStoreService` object using a new operator, we will instruct Angular to create an instance of service and inject it into the component, this a two-step process:

- Pass the service object to the component `constructor` as a parameter
- Specify the service to which we need an object in the providers array of the `@Component({ providers: [] })` decorator on the component:

```
import {BookStoreService} from './contacts/contacts.service';

@Component({
    providers: [BookStoreService]
})
export class AppComponent {
    constructor(private bookStoreService:BookStoreService) { }
}
```

In the preceding code snippet, when Angular looks at the constructor parameter, it will go to the `providers` array in the decorator for a matching provider, and it creates an instance of the mentioned service using the provider. We will learn in detail about why this is happening and why we need to specify the service in `providers` arrays.

The preceding code snippet does a little bit of magic because of TypeScript. For any public or private parameters mentioned on the `constructor`, TypeScript automatically creates a property in the class and assigns it with the value of the `constructor` parameter inside the `constructor`. TypeScript interprets the preceding code like this:

```
class AppComponent {
    bookStoreService: BookStoreService;

    constructor(private bookStoreService: BookStoreService) {
        this.bookStoreService = bookStoreService;
    }
}
```

The following is the full rewritten implementation of `AppComponent` using `BookStoreService`.

The code for `src/app.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Book } from './book';
```

```
import { BookStoreService } from './book-store.service';

@Component({
  selector: 'books-list',
  templateUrl: 'src/app.component.html',
  providers: [BookStoreService]
})
export class AppComponent implements OnInit {

  booksList: Book[];
  selectedBook: Book;

  constructor(private bookStoreService: BookStoreService) { }

  ngOnInit() {
    this.getBooksList();
  }

  getBooksList() {
    this.booksList = this.bookStoreService.getBooks();
  }

  getBookDetails(isbn: number) {
    this.selectedBook = this.bookStoreService.getBook(isbn);
  }

  deleteBook(isbn: number) {
    this.selectedBook = null;
    this.booksList = this.bookStoreService.deleteBook(isbn);
  }
}
```

Here are a few important points about services:

- Services declared inside the `providers` array of a parent component are available to child components out of the box
- Child components do not need to declare the services again in their `providers` array
- If a child component declares a service again in its `providers` array, which is already declared in its parent component providers array, Angular will create a new instance of service for the child component, and it will not use the parent component's service
- A service instance created at a child component is accessible only to itself and its child components

- Instead of declaring services in the `providers` array of a component, we can also declare them in the providers array of the module using a `@NgModule({ providers: [] })` decorator
- Services declared at module level are available throughout the module and its submodules

One important thing in the `AppComponent` class is that instead of calling the `getBooksList()` method directly in the `constructor`, we are calling it a special method called `ngOnInit()`. As mentioned earlier, the `ngOnInit()` method is a component lifecycle hook method invoked right after the component is created. The `constructor` job is just to build and initialize the object, not fetching the data, this is taken care of by the `ngOnInit()` method.

Dependency injection

In the previous section, we learned what dependency injection is and the need for it. We skipped a few things like:

- How is a service object created and injected into the `constructor`?
- Why do we need to specify the service objects in the `providers` array?
- Different mechanisms for creating a provider.

Using a class provider

As mentioned in the previous section, getting the instance of a service object is a two-step process. In step one, we pass the service object as a parameter to the `constructor`:

```
export class AppComponent {
  constructor (private bookStoreService: BookStoreService) {
  }
}
```

At this moment, Angular does not know how to create an instance of `BookStoreService`, the instance creation process is specified in the `providers` array in the decorator of the `Component` class. The following code snippet is for our service from the previous example:

```
@Component({
  providers: [BookStoreService]
})
export class AppComponent {
```

```
    constructor(private bookStoreService: BookStoreService) {
    }
}
```

In the preceding code snippet, in the decorator `providers` array, we simply gave the same service name, `providers: [BookStoreService]`, which is passed as a constructor parameter.

How does the `providers` array create an object for `BookStoreService`?

```
    providers: [BookStoreService]
```

Well, the preceding code in the `providers` array is shorthand syntax. Here is how Angular interprets it:

```
    [{ provide: BookStoreService, useClass: BookStoreService }]
```

The preceding code snippet is an expanded version of a `providers` array; it is an object literal with two properties:

- The first property, `provide` is a token that serves as the key for both registering a dependency value with an injector object and locating the provider from injector object
- The second property, `useClass` is a strategy used for creating the actual provider definition object, which is the dependency value
- There are many ways to create dependency values; `useClass` is one of them

In our case, both the key (token) and value (provider definition object) are the same, shorthand syntax can be used only in this scenario.

Using a class provider with dependencies

Most of the time, our service depends on other services, and we inject those services into our service constructor. However, we need to inform Angular how to create instances of our dependencies. Here we have a service named `ConsoleLoggerService` which logs data into the console:

```
@Injectable()
export class ConsoleLoggerService {
  log (message: string) {
    console.log(message);
  }
}
```

Our `BookStoreService` is using a `ConsoleLoggerService` service to log the data, and it is injected into the `BookStoreService` constructor:

```
@Injectable()
export class BookStoreService {

  constructor (private loggerService: ConsoleLoggerService)
  {}

  getBook (isbn: number) {
    this.loggerService.log('fetching book information');
  }
}
```

We used our `BookStoreService` in the component and mentioned it in the `providers` array, but now `BookStoreService` is dependent on `ConsoleLoggerService` which is a class. We can also simply specify it in the `providers` array, and it works.

In the following sections, we will learn how to deal with non-class dependencies like interfaces and strings.

Using alternate class providers

The current `BookStoreService` retrieves all the data from a dummy data source. In the future, we may decide to use GraphQL or a different implementation for our data source.

For example, suppose we implement a new service called `BookStoreGraphQLService`, and this service also provides the same API as `BookStoreService`, we can simply swap our `BookStoreService` provider with `BookStoreGraphQLService`:

```
providers: [{
  provide: BookStoreService,
  useClass: BookStoreGraphQLService
}]
```

Now, for all components where the `BookStoreService` key is injected, they will use the `BookStoreGraphQLService` instance.

Using aliased class providers

Here we have a different scenario. In the future, we will implemented new `BookStoreGraphQLService`. We decided that all the new components will use this implementation and old components continuously to use the existing `BookStoreService` implementation. We can register the new service in the `providers` array and use it as usual:

```
providers: [BookStoreGraphQLService, BookStoreService]
```

Though we have two different service applications working nicely, some day we may decided that all the old components should also use the new `BookStoreGraphQLService` service. One way is to go to all the components and services where the `BookStoreService` key is used and replace it with the `BookStoreGraphQLService` key, which is not a good option. Instead of modifying in all those places, we can specify that the `BookStoreService` key use the `BookStoreGraphQLService` provider object using the `useClass` strategy:

```
providers: [ BookStoreGraphQLService,
             {
                provide: BookStoreService,
                useClass: BookStoreGraphQLService
             }
           ]
```

The old components also now use the new `BookStoreGraphQLService`. However, there is a small problem, if we look at how Angular interprets the preceding code:

```
providers: [ {
                provide: BookStoreGraphQLService,
                useClass: BookStoreGraphQLService
             },
             {
                provide: BookStoreService,
                useClass: BookStoreGraphQLService
              }
            ]
```

The Angular `useClass` strategy always creates a new instance of a given `provider` service class, so here we will have two instances of `BookStoreGraphQLService` instead of one, which is unnecessary. We can instruct Angular to use the existing `BookStoreGraphQLService` instance for different tokens (provide) using the `useExisting` strategy:

```
providers: [ BookStoreGraphQLService,
             {
                 provide: BookStoreService,
                 useExisting: BookStoreGraphQLService
             }
           ]
```

There are two more different kinds of provider instantiation strategies; factory and value providers, for creating the instance of a provide service object, which we will discuss in future chapters.

Summary

We started this chapter by discussing the book list application which we built in previous chapters. Then discussed how to break the single component into multiple components and how the components communicate with each other using input and output properties. Then discussed how to build a common data access point for components using services to share the data between them. Finally, we discussed different strategies used for creating an instance of provider service objects.

By the end of this chapter, the reader should have a good understanding of how to build any UI application using multiple components and how to share the data between them. In the next chapter, we will discuss how to create applications using RxJS and observables.

4
Working with Observables

In this chapter, we are going to look at the reactive programming paradigm embraced by Angular and focus on how data flows through an application. We use Observables to implement reactive programming concepts. ES7 has a proposal to include Observables into JavaScript language. Today we can use them with the **Reactive-Extensions for JavaScript** (**RxJS**) library. This chapter will cover only the essential concepts of RxJS, and there are a good number of resources available for learning RxJS mentioned at the end. After going through this chapter, we will understand the following concepts:

- Reactive programming
- RxJS basics
- What are Observables and operators?
- Writing components and services using Observables

Basics of RxJS and Observables

Before getting started with the basics of RxJS and Observables, first we need to understand what reactive programming is and why it is important.

Reactive programming

In traditional imperative programming, a variable state will be modified when we are explicitly assigning a new or updated value. In this case, the variable will lose its previous value; here data is propagated using a pull mechanism, any part of an application dependent on this variable or object has to pull the value explicitly when there are changes, they are not propagated automatically.

Reactive programs work in an opposite fashion. Instead of explicitly assigning new values, they are pushed implicitly, and changes are propagated automatically to all the dependent parts of the application. We will learn how to write reactive programs using Observables in coming sections.

To learn more about reactive programming, check out the following links:

- `https://en.wikipedia.org/wiki/Reactive_programming`
- `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`

Let us learn the basics of RxJS, Observables, and operators.

Observer

The `Observer` is collection callbacks that know how to listen to values emitted by an Observable:

```
interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

The `Observer` object has three callback methods: `next()`, `error()`, and `complete()`. These methods are explained in detail, as follows:

- Every time an Observable emits the value, the `next()` callback is invoked
- If there no more values emitted by Observable, the `complete()` callback is invoked
- The `error()` callback will be invoked if an error is thrown, then the `Observer` will stop listening to values

Observable

The Observable is a collection of values or events that arrive over time; it can model events, asynchronous server requests, or animations in the UI. The `Observable` class has many methods for creating Observable collections:

- `Observable.create()`
- `Observable.of()`
- `Observable.from()`
- `Observable.fromArray()`
- `Observable.fromEvent()`
- `Observable.fromPromise()`
- `Observable.interval()`
- `Observable.timer()`

The Observable is the core building block of RxJS; it is important for us to understand how to use it in Angular and how Angular uses it internally. Let us use the `Observable.create()` method to create it manually. The following example demonstrates some of the key concepts of Observable:

The code for `example01.html` is as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Manually creating an Observable</title>
</head>
<body>
<script type="text/babel">
  const observable = Rx.Observable.create((observer) => {
    observer.next(1);
    observer.next(2);

    setTimeout(() => {
      observer.next(3);
      observer.next(4);
      observer.complete();
    }, 1000);

    observer.next(5);
  });
```

```
   console.log('Before subscribe');

   observable.subscribe({
     next: val => console.log(`Got value ${val}`),
     error: err => console.log(`Something went wrong ${err}`),
     complete: () => console.log('I am done')
   });

   console.log('After subscribe');
</script>

<script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
<script src="https://unpkg.com/@reactivex/rxjs/dist/global/Rx.js">
</script>

</body>
</html>
```

If we open the `example01.html` file in the browser, we can view the following output messages in the browser console:

```
Before subscribe
Got value 1
Got value 2
Got value 5
After subscribe
Got value 3
Got value 4
I am done
```

The `Observable.create()` method creates a new Observable using an `Observer`. The Observable will emit values only when an `Observer` subscribes to it using the `subscribe()` method, we can clear understand this behavior viewing at output messages.

In the preceding example, we saw the `Before subscribe` message first, even though the `Observable` object is already created, it will emit the values only after `subscribe()` method is invoked.

Here is an example of working with DOM events using an Observable.

The code for `example02.html` is as follows:

```
const mouseMoves = Rx.Observable.fromEvent(document, 'mousemove');

mouseMoves
  .subscribe(event => console.log(event.clientX, event.clientY));
```

The preceding example will log all the mouse movement to the browser's console.

Subscription

The `subscription` object represents the execution of an Observable, and it is used for canceling the execution.

The code for `example03.html` is as follows:

```
const interval = Rx.Observable.interval(1000);
const subscription = interval.subscribe(val => console.log(val));

setTimeout(() => {
  subscription.unsubscribe();
}, 10000);
```

In the preceding example, Observable emits a value every one second, and we are logging to the browser console. The Observable stops emitting the values after ten seconds, because we are unsubscribing from it using the `unsubscribe()` method on the `Subscription` object returned by the `subscribe()` method.

Operators

An operator is a pure function which creates a new Observable based on the current Observable, and lets us perform various kinds of operations like filtering, mapping, and delaying values. RxJS is very rich in terms of operators, and throughout the chapter we will learn different types of operators.

Here is an example using `map()` and `filter()` operators.

The code for `example04.html` is as follows:

```
const interval = Rx.Observable.interval(1000)
                    .map(x => x * 2)
                    .filter(x => x%2 === 0);

interval.subscribe(val => console.log(val));
```

In the preceding example, we use `map()` operator to multiply the values, then we use the `filter()` operator to filter even values.

Observables in Angular

Angular uses Observables internally in a lot of concepts like forms, HTTP, and router. In this chapter, we will only look at how to use Observables with events and how to use operators.

Observable stream and mapping values

Here is an example of handling a button click and textbox input using an Observable.

The code for `example05/src/app.component.ts` is as follows:

```
import { Component, ElementRef, OnInit, ViewChild } from
  '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  template: `
    <div class="container">
      <input #text class="form-control mt-1"/>
      <button #btn class="btn primary mt-1">Show Message!</button>
      <p class="mt-1">{{message}}</p>
    </div>
  `
})
export class AppComponent implements OnInit {
  @ViewChild('btn') btn;
  @ViewChild('text') text;
```

```
  message: string;

  ngOnInit() {
    const btnOb$ = Observable
.fromEvent(this.btn.nativeElement, 'click');
    btnOb$
      .subscribe(res => this.message = 'Hello Angular, RxJS!');

    const textOb$ = Observable
.fromEvent(this.text.nativeElement, 'change')
.map((event: Event) => (<HTMLInputElement>event.target).value);
    textOb$.subscribe(res => this.message = res);
  }
}
```

Let us understand what is happening in the preceding `Component`:

- We are accessing a button and textbox which are in the `template` using `@ViewChild` in `Component`
- We are accessing the underlying DOM elements using `nativeElement` property
- We are creating an Observable, one for the button click and another for the text change event
- When the button is clicked, we are displaying the `'Hello Angular, RxJS!'` message

- When the textbox text is changed, we are displaying the same text in the message

Merging Observable streams

In the preceding example, we have two redundant subscribe blocks doing the same thing, we can refactor them using the `merge()` operator. We can include the `merge()` operator using `import 'rxjs/add/operator/merge'`:

```
ngOnInit() {
  const btnOb$ = Observable
                  .fromEvent(this.btn.nativeElement, 'click')
    .map(event => 'Hello Angular, RxJS!');

  const textOb$ = Observable
.fromEvent(this.text.nativeElement, 'change')
    .map(event => event.target.value);


  Observable
```

```
.merge(btnOb$, textOb$)
.subscribe(res => this.message = res);
}
```

We are using the `merge()` operator to combine both streams and subscribing to the output stream, and it will concurrently emit all values from every given input Observable. In our case, either the user clicks on the button or enters text into a textbox and we are going to display the following message:
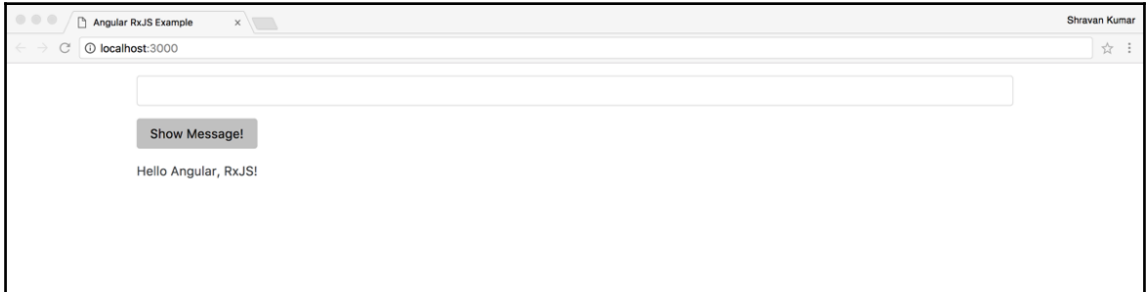
Using the Observable.interval() method

Let us build one more example to display a clock and understand some more concepts.

The code for `example06/src/app.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  template: `
    <div class="container">
```

```
        <p class="mt-1">{{time}}</p>
      </div>
    `
})
export class AppComponent implements OnInit {
  time: string;

  ngOnInit() {
    const timer$ = Observable.interval(1000)
      .map(event => new Date());

    timer$.subscribe(val => this.time = val.toString());
  }
}
```


The preceding example updates the timer on the view every one second to display the clock. Instead of subscribing to the `timer$` Observable, let us display it directly in views:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  template: `
    <div class="container">
      <h4 class="mt-1">{{timer$}}</h4>
    </div>
  `
})
export class AppComponent {

  timer$ = Observable.interval(1000)
    .map(event => new Date());
}
```

The preceding code snippet will display `[object Object]` on the browser screen. Because `timer$` Observable is an object not a value, but the `timer$` Observable emits the date and time.

We can access this value only the `subscribe()` method. Angular provides the `AsyncPipe` to access the values emitted by an Observable directly in the view.
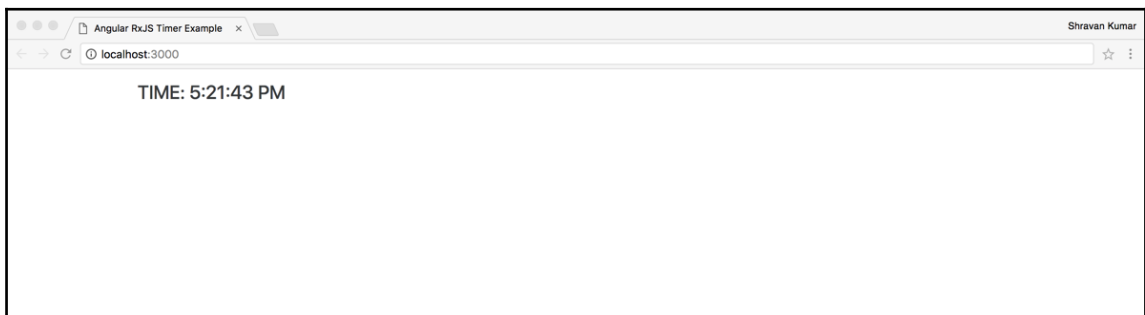
Using AsyncPipe

The `async` pipe subscribes to an Observable internally and returns the latest value it has emitted:

```
template: `
  <div class="container">
    <h4 class="mt-1">{{timer$ | async}}</h4>
  </div>
`
```

Now we get the same output as previously but without directly subscribing to the Observable. Let us format our date using DatePipe just to display only the time:

```
template: `
  <div class="container">
    <h4 class="mt-1">
      TIME: {{timer$ | async | date: 'mediumTime'}}
    </h4>
  </div>
```

Building a Books Search component

To understand Observables in depth, we are going to look at one more example. In `Chapter 2`, *Basics of Components*, we built a `master-details` books application. Let us add search functionality to it. We need the following functionality in the search form:

- When a user starts typing in the search box, we should show the book title suggestion
- The user should be able to select the title from the suggestions
- The user should be able to search and see a list of books based on the input entered in the search box

> All the required source code for setup is available under `chaper04/books-search` in provided source code.

The following is `BookSearchComponent` where we are going to implement the search functionality using Observables and RxJS operators.

The code for `src/books/book-search/book-search.component.ts` is as follows:

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';

@Component({
  moduleId: module.id,
  selector: 'book-search',
  styleUrls: ['./book-search.component.css'],
  template: `
    <h3 class="page-title">Books Search</h3>
    <div class="search-container">
      <div class="books-search-form">
        <input type="text" #searchInput
                class="search-input" placeholder="Book Title">
        <button class="btn btn-primary">Search</button>
      </div>
      <ul>
        <li *ngFor="let bookTitle of bookTitles">
         {{bookTitle}}
</li>
      </ul>
    </div>
```
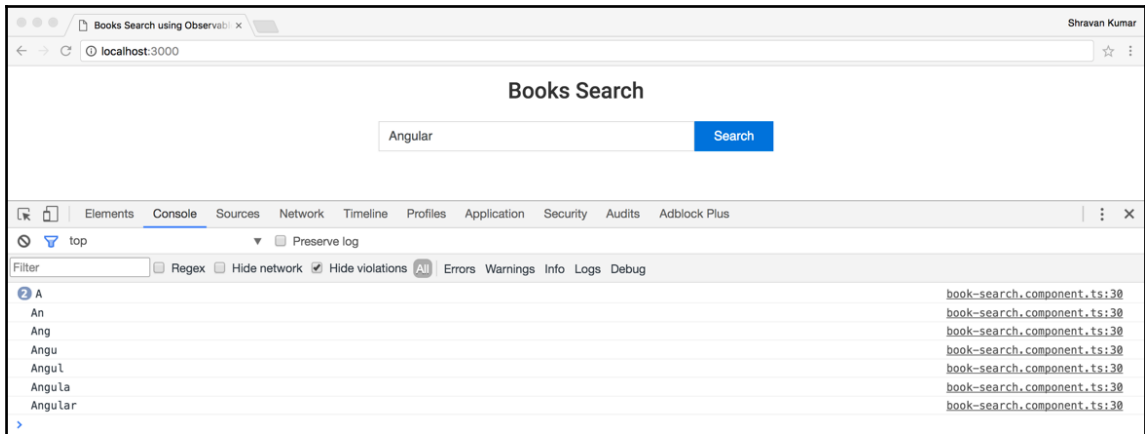
```
        `
      })
      export class BookSearchComponent implements OnInit {
        @ViewChild('searchInput') searchInput;
        bookTitles: Array<string>;

        ngOnInit() {
          Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
            .map((event: KeyboardEvent) =>
                                (<HTMLInputElement>event.target).value)
            .subscribe(title => console.log(title));
        }
      }
```


In the preceding `Component`, we are capturing all user input entered into the search box using an Observable and displaying it in the console.

To search book titles and books based on user input, we need to implement that functionality in a service, following `BookStoreService` implementing that. It has two methods, one for searching books and one for searching book titles.

The code for `src/books/book-store.service.ts` is as follows:

```
      import { Injectable } from '@angular/core';
      import { Observable } from 'rxjs/Observable';
      import 'rxjs/add/observable/of';

      import { Book } from './book';
      import MOCK_BOOKS from './mock-books';

      @Injectable()
```

```
export class BookStoreService {

  booksList: Book[] = MOCK_BOOKS;

  getBooks(title: string): Observable<Book[]> {
    return Observable.of(this.filterBooks(title));
  }

  getBookTitles(title: string): Observable<string[]> {
    return Observable.of(this.filterBooks(title)
                      .map(book => book.title));
  }

  filterBooks(title: string): Book[] {
   return title ?
      this.booksList.filter((book)
        => new RegExp(title, 'gi').test(book.title)) :
    [];
}
}
```

We can update our search component to use `BookStoreService` for the book title suggestions when the user starts entering input.

The code for `src/books/book-search/book-search.component.ts` is as follows:

```
ngOnInit() {
  Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
    .map((event: KeyboardEvent) =>
                        (<HTMLInputElement>event.target).value)
    .subscribe(title =>
      this.bookStoreService
          .getBookTitles(title)
          .subscribe(bookTitles => this.bookTitles = bookTitles));
}
```
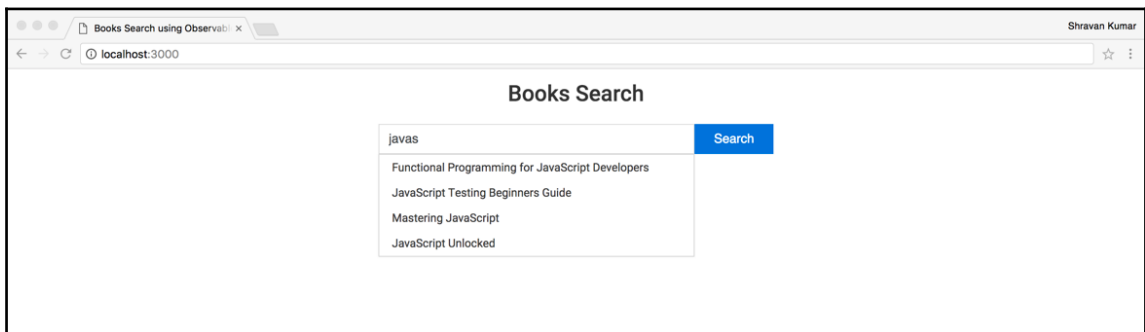
In the `subscribe()` method, we are calling the `getBookTitles()` method and passing text entered in the search box, which again returns the book title results Observable.

Everything looks nice; we are getting the results, but there is something that is not right in the preceding code snippet. We are using one `subscribe()` inside another `subscribe()` method this again, is similar to nested callbacks. We should not write the code this way using RxJS. To deal with this kind of problem, RxJS provides many operators.

In our case, we can use the `mergeMap()` operator; it takes the source value of the input Observable and produces a flat output Observable based on applying a function that we provide.

The code for `src/books/book-search/book-search.component.ts` is as follows:

```
ngOnInit() {
  Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
    .map((event: KeyboardEvent) =>
                         (<HTMLInputElement>event.target).value)
    .mergeMap(title => this.bookStoreService.getBookTitles(title))
    .subscribe(bookTitles => this.bookTitles = bookTitles);
}
```

We need to refactor this code to perform better. Right now, as soon as the user starts typing, we are making services calls. The application should wait for the user to enter some characters and only then make the services call, we also do not need to call the service again if the next search term is the same as previous. This can be achieved using `debounceTime()` and `distinctUntilChanged()` operators.

The code for `src/books/book-search/book-search.component.ts` is as follows:

```
ngOnInit() {
  Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
    .debounceTime(400)
    .distinctUntilChanged()
    .map((event: KeyboardEvent) =>
                    (<HTMLInputElement>event.target).value)
    .switchMap(title =>
               this.bookStoreService.getBookTitles(title))
    .subscribe(bookTitles => this.bookTitles = bookTitles);
}
```

The `debounceTime(400)` operator waits for 400 ms after each keystroke before considering the search term, the `distinctUntilChanged()` operator ignores it if the next search term is the same as previous. Now we are using the `switchMap()` operator instead of the `mergeMap()` operator; it switches to a new Observable each time the search term changes.

We make more changes to our code to send the search term to the parent component when a user clicks on the search button. We will do that using the @Output() decorator, as we learned in the last chapter.

The code for src/books/book-search/book-search.component.ts is as follows:

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { Output, EventEmitter } from '@angular/core';
import { BookStoreService } from '../book-store.service';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
  moduleId: module.id,
  selector: 'book-search',
  templateUrl: './book-search.component.html',
  styleUrls: ['./book-search.component.css']
})
export class BookSearchComponent implements OnInit {
  @ViewChild('searchInput') searchInput;
  @ViewChild('suggestions') suggestions;
  bookTitles: Array<string> = [];
  searchInputTerm: string = '';

  @Output() search = new EventEmitter<string>();

  constructor(private bookStoreService: BookStoreService) {
  }

  ngOnInit() {
    Observable.fromEvent(this.searchInput.nativeElement, 'keyup')
      .debounceTime(400)
      .distinctUntilChanged()
      .map((event: KeyboardEvent) =>
                          (<HTMLInputElement>event.target).value)
      .switchMap(title =>
                    this.bookStoreService.getBookTitles(title))
      .subscribe(bookTitles => this.bookTitles = bookTitles);

    Observable.fromEvent(this.suggestions.nativeElement, 'click')
      .map((event: KeyboardEvent) =>
                  (<HTMLInputElement>event.srcElement).innerText)
      .subscribe(res => {
```

```
        this.searchInputTerm = res;
        this.bookTitles = [];
      });
  }

  searchBooks() {
    this.bookTitles = [];
    this.search.emit(this.searchInputTerm);
  }
}
```

The code for `src/books/book-search/book-search.component.html` is as follows:

```html
<h3 class="page-title">Books Search</h3>
<div class="search-container">
  <div class="books-search-form">
    <input type="text" #searchInput class="search-input"
           placeholder="Book Title" [(ngModel)]="searchInputTerm">
    <button class="btn btn-primary" (click)="searchBooks()">
Search
    </button>
  </div>
  <div class="title-suggestion-list--wrapper">
    <ul class="title-suggestion-list" #suggestions
[style.display]="bookTitles.length > 0 ? 'block' : 'none'">
      <li *ngFor="let bookTitle of bookTitles">{{bookTitle}}</li>
    </ul>
  </div>
</div>
```

Here is the `AppComponent` hosting `BookSearchComponent` and `BooksListComponent` which is the root component of our application.

The code for `src/app.component.ts` is as follows:

```typescript
import { Component } from '@angular/core';
import { BookStoreService, Book } from './books/index';

@Component({
  selector: 'app-root',
  template: `
    <div class="container">
      <book-search (search)="searchBook($event)"></book-search>
      <books-list [books]="filteredBooks"></books-list>
    </div>
  `,
  providers: [BookStoreService]
})
```

```
export class AppComponent {
  filteredBooks: Book[];

  constructor(private bookStoreService: BookStoreService) {
  }

  searchBook(title: string) {
    this.bookStoreService
      .getBooks(title)
      .subscribe(books => this.filteredBooks = books);
  }
}
```

Below is the `BooksListComponent` used in `AppComponent` to display a book list based on user search input.

The code for `src/books/books-list/books-list.component.ts` is as follows:

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

@Component({
  selector: 'books-list',
  styles: [`
    .book-item {
      margin-bottom: 1rem;
    }
.cover-image-container {
      width: 100%;
    }
    .cover-image-container img {
      width: 100%;
      vertical-align: 0;
      border: 0;
    }
  `],
  template: `
    <div class="row mt-1">
      <div class="col-sm-12">
        <div class="row">
          <div class="col-sm-3 book-item"
              *ngFor="let book of books">
            <div class="cover-image-container">
              <img [src]="book.coverImage" alt="cover image"/>
            </div>
          </div>
        </div>
      </div>
```

```
      </div>
    </div>
    `
})
export class BooksListComponent {
  @Input() books: Book[] = [];
}
```

Here is the final output of our application:
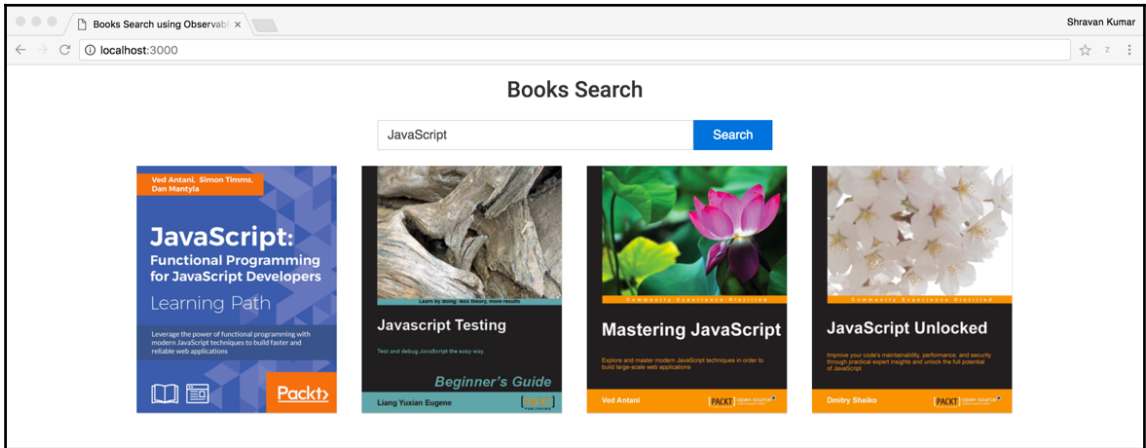
> All the preceding example's source code is available under *chaper4* in the provided source code.

As mentioned at the start of this section, we have different concepts like the router module, forms module, and HTTP module that are implemented using Observables. We will continue learning how to use Observables in coming chapters.

> More resources for learning RxJS can be found at `http://reactivex.io/rxjs/`

Summary

We started this chapter with what reactive programming is and how to implement it using the concept of Observables. Next, we looked at the basics of RxJS, such as Observables and operators and how to use them to write Angular components and services in different scenarios.

By the end of this chapter, the reader should have a good understanding of different RxJS concepts like what Observables and operators are, and how to use them in various scenarios. In the next chapter, we will discuss how to build forms using Angular.

5

Handling Forms

In this chapter, we are going to learn how to use the new forms API in Angular to build user interfaces to capture, validate, and submit user inputs. After going through this chapter, the reader will understand the following concepts:

- Template driven forms in Angular
- Reactive forms in Anlgular
- Validating form inputs

Why are forms hard?

Forms are the key to any web application; they help us in capturing input from users. Here are a couple of things we do with forms:

- Capturing input from the user
- Validating the user input
- Responding to events
- Displaying the information messages
- Displaying the error messages

Validations make forms harder to deal with because we do not know in which fashion the user enters the data. One control logic might be dependent on other control input. Sometimes, we need to trigger validation logic on the server based on user input (checking uniqueness of username or e-mail address). We need to maintain the overall state of the form even if it spans multiple templates, like wizards. Angular provides a simpler approach for capturing the user input as well as for dealing with validations.

Angular forms API

On the DOM, we have input controls, and we need information about the controls, such as their value, whether the data entered is valid according to validation rules, how the user has interacted with the control, whether they changed its value or touched it yet, and how we want to be notified of their events (click, blur, and other DOM events) when they occur. Angular has the following two approaches for dealing with forms:

- Template driven forms
- Reactive forms

Each technique has different opinions on how to handle forms; we will look at them in detail in upcoming sections.

FormControl, FormGroup, and FormArray

The `FormControl, FormGroup,` and `FromArray` classes are the key to both techniques. Let us understand these classes first, and then we can explore each method in detail.

FormControl

Control is the smallest unit in any form; it represents a single form input element (textbox, dropdown, radio button, checkbox, and so on). Control is the fundamental building block of forms API in Angular; a control object encapsulates the input field's value and its state. It is represented using the `FormControl` class.

Creating a form control

The following code snippet creates a single control named `firstName`:

```
let firstName = new FormControl();
```

The following code snippet creates a single control named `firstName` and initializes it with an empty default value:

```
let firstName = new FormControl('');
```

The following code snippet creates a single control named `firstName`, and initializes with default value `'Shravan'`:

```
let firstName = new FormControl('Shravan');
```

Accessing the value of an input control

Using value property of form control object, we can get the value of the input:

```
let firstNameValue = firstName.value;
```

Setting the value of input control

We cannot use value property to set the value of form control; it is just a getter. We should use the setValue() method to set the value programmatically:

```
firstName.setValue('Shravan');
```

Resetting the value of an input control

The reset() method on the form control sets the value to null:

```
firstName.reset();
```

Input control states

Every input control on an Angular form and the form itself maintains different states depending on the user input and interaction with it:

```
//form control error list object
let errors = firstName.errors

// form control value is valid, it has no errors
let isValid = firstName.valid

// form control value is invalid, it has errors
let isInValid = firstName.invalid

//Control has been visited
let isTouched = firstName.touched

//Control has not been visited
let isUnTouched = firstName.untouched

//Form control's value has changed
let valueChanged = firstName.dirty

//Form control's value has not changed
let valueNotChanged = firstName.pristine
```

Whenever an input control's state changes, Angular will update the element with the following classes:

State	Class if true	Class if false
Control has been visited	`ng-touched`	`ng-untouched`
Control's value has changed	`ng-dirty`	`ng-pristine`
Control's value is valid	`ng-valid`	`ng-invalid`

The preceding states and classes are not applicable to only the `FormControl` object, they shall also apply to `FormGroup`, `FormArray`, and the entire form.

FormGroup

Even a simple form contains more than one control that might be dependent on each other. Instated of working with each control and iterating over them to know the value and state of each control and form, we want to know the state of multiple controls at once. Sometimes it makes more sense to think of a series of form controls as a group.

We have another class, `FormGroup` which comes in handy. It is a collection of form controls and maintains the overall state of the form. For example, we need a user address which contains the street, city, state, country, and zip code. We can create five individual `FormControl` objects and work them one at a time, but all together they represent an address where we can use the `FormGroup` class:

```
//create a form group
let address = new FormGroup({
  street: new FormControl(''),
  city: new FormControl(''),
  state: new FormControl(''),
  country: new FormControl(''),
  zip: new FormControl('')
});

//return an object literal of form group value
let formModel = address.value; //{street: "", city: "", state: "",
country: "", zip: ""}

//check overall state form state
let errors = address.errors; //null
let isValid = address.valid; //true
let isInValid = address.invalid; //false
let isTouched = address.touched; //false
```

```
let isUnTouched = address.untouched; //true
let valueChanged = address.dirty; //false
let valueNotChanged = address.pristine; //true

//set the value of the form group
address.setValue({
  street: '1-3 Strand',
  city: 'London',
  state: '',
  country: 'UK',
  zip: 'WC2N 5BW'
});
```

We can use the `setValue()` method to set the value for `FormGroup` programmatically, but we must pass all the controls which are declared initially with `FormGroup`. Otherwise, the `setValue()` method throws an error.

If we need to update `FormGroup` partially from a superset or subset, we can use the `patchValue()` method:

```
address.patchValue({
  street: '1-3 Strand',
  city: 'London'
});
```

The `patchValue()` method accepts both supersets and subsets of the group without throwing an error.

> Form (`<form></form>`) itself is represented using the `FormGroup` class.

FormArray

The `FormArray` class is similar to the `FormGroup` class, it is also a collection of form controls and maintains the overall state of the form. We can use `FormArray` to create a form of variable or unknown length:

```
//create a form array
let registration = new FormArray([
  new FormControl('Shravan'),
  new FormControl('Kasagoni'),
  new FormControl('shravan@theshravan.net')
]);
```

```
registration.push(new FormControl('UK'));
registration.patchValue(['London','W5']);

//access form array value
console.log(registration.value);
console.log(registration.value[0]);
```

Now, we have learned the foundation classes for the Angular forms module. Let us dive into different approaches provided by Angular.

Template driven forms

The template driven forms approach is similar to working with forms in Angular 1.x. As the names suggest, we will write all the logic, like creating form controls, forms, and defining validations inside the template in a declarative manner.

Creating a registration form

To begin with template driven forms in Angular, let us start with creating a project named `forms` and using the following directory structure and files:

```
forms
├── index.html
├── package.json
├── src
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── main.ts
│   └── registration-form
│       ├── registration-form.component.html
│       └── registration-form.component.ts
├── styles.css
├── systemjs-angular-loader.js
├── systemjs.config.js
└── tsconfig.json
```

We need to add the code to `package.json`, `tsconfig.json`, `systemjs-angular-loader.js`, `system.config.js`, and `index.html` from the last example in `Chapter 4`, *Working with Observables*.

Before we run the application, let us make sure we have `"@angular/forms": "^4.0.0"` added to the dependencies section in the `package.json` file and add the following line; `'@angular/forms': ng:forms/bundles/forms.umd.js'`, to map the object in the `systemjs.config.js` file.

The code for `styles.css` is as follows:

```
/**
The stylesheet is very long, reader can add it from sample code under
chapter5/forms example.
**/
```

The code for `src/app.module.ts` is as follows:

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

import { AppComponent } from './app.component';
import { RegistrationFormComponent } from './registration-
form/registration-form.component';

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

We added `FormsModule` from the `'@angular/forms'` package to `import` arrays, all the classes related to template driven forms are in this module. The `RegistrationFormComponent` is added to the `declarations` array, and this component is part of our `AppModule`, we can access it anywhere in our `AppModule`.

The code for `src/app.component.ts` is as follows:

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

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

We do not have much code in `AppComponent`, it is just a placeholder for displaying `RegistrationFormComponent`. In the `AppComponent` template, we are using the `<registration-form>` tag which is the selector for `RegistrationFormComponent`. Also, make sure we update the `<forms-app>` selector inside the `<body>` tags in `index.html`.

All the upcoming examples will use the `AppComponent` just as a placeholder for displaying other components, there is no added advantage here but we will understand why we are doing this in future chapters.

The code for `src/registration-form/registration-form.component.ts` is as follows:

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

@Component({
  selector: 'registration-form',
  templateUrl: './registration-form.component.html'
})
export class RegistrationFormComponent {
}
```

The code for `src/registration-form/registration-form.component.html` is as follows:

```
<div class="row m-1">
    <div class="col-md-8">
        <div class="box">
            <div class="box-header">
                <h2>Registration Form</h2>
            </div>
            <div class="box-divider"></div>
            <div class="box-body">
                <div class="row">
                    <div class="col-sm-6 form-group">
                        <label>First name</label>
                        <input type="text" class="form-control">
                    </div>
                    <div class="col-sm-6 form-group">
                        <label>Last name</label>
                        <input type="text" class="form-control">
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

[98]

It looks like there is a lot of code in the template, but we have only two textboxes; the other pieces are just HTML and CSS classes from Bootstrap for styling purposes. There is no Angular forms-related code yet.

Our application is ready. Now run the `npm install` command, once it is finished run the `npm start` command. This will start our application in the browser. We can view the following output in the browser:
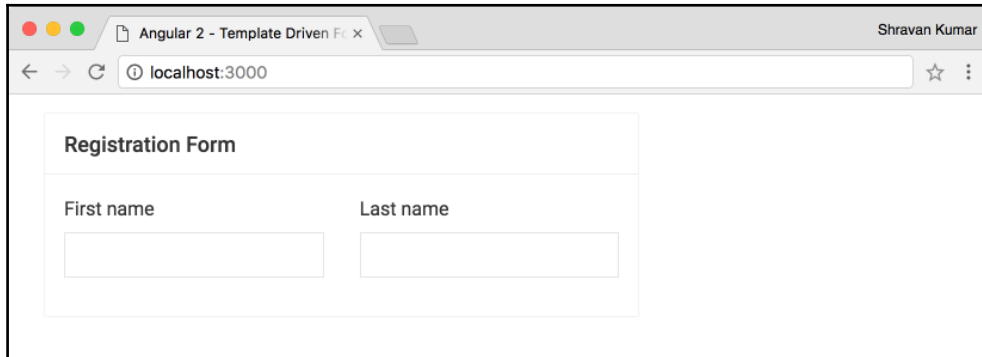
We have two HTML input controls on the page. We need to make them form controls by adding some Angular forms-related code.

When we are working with template driven forms, we will never directly create `FormControl`, `FormGroup` object on our own. Instead we will use `ngModel`, `ngModelGroup`, `ngForm` directives on input controls everything internally handled by Angular.

Using the ngModel directive

To work with individual input controls, we should use the `ngModel` directive. Let us add it to our input controls (`First name` and `Last name`) in the `registration-form.component.html` template:

```
<input type="text" class="form-control" ngModel>

<input type="text" class="form-control" ngModel>
```

We added `ngModel` to our input controls. There won't be any change in browser output, but under the hood Angular creates two `FormControl` objects for the `First name` and `Last name`.

Accessing an input control value using ngModel

The `ngModel` directive on the input controls represents the model object. To access it we need to export to a template reference variable:

```
<input type="text" class="form-control"
ngModel #firstNameRef="ngModel">

<input type="text" class="form-control"
ngModel #lastNameRef="ngModel">
```

Now, we can use these `#firstNameRef` and `#lastNameRef` template reference variables to access the model (`FormControl`) objects of the `First name` and `Last name` input controls anywhere in the template. Let us use the template reference variables and interpolation syntax to display the text typed into input controls:

```
<input type="text" class="form-control"
ngModel #firstNameRef="ngModel">
{{firstNameRef}}

<input type="text" class="form-control"
ngModel #lastNameRef="ngModel">
{{lastNameRef}}
```

Once we save the template, the browser will refresh with the following output:
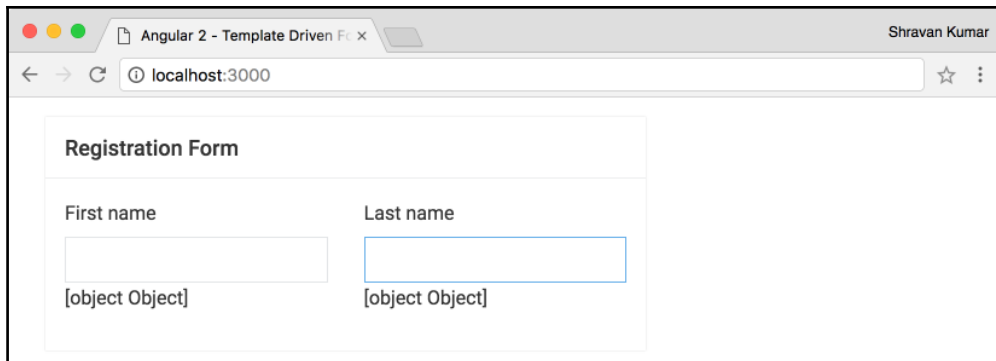
The interpolation is displaying objects because, as mentioned earlier, template reference variables to exported with the model(`FormControl`) object of input controls. We should use properties to access their values and states:

```
{{firstNameRef.value}}

{{lastNameRef.value}}
```

After saving the code, once the browser refreshes, starting typing the same text in any of the input fields. Immediately we can see those values getting displayed on the browser screen:
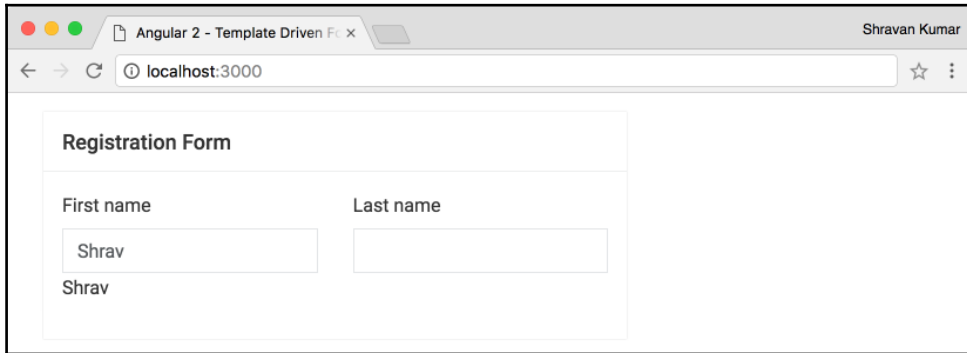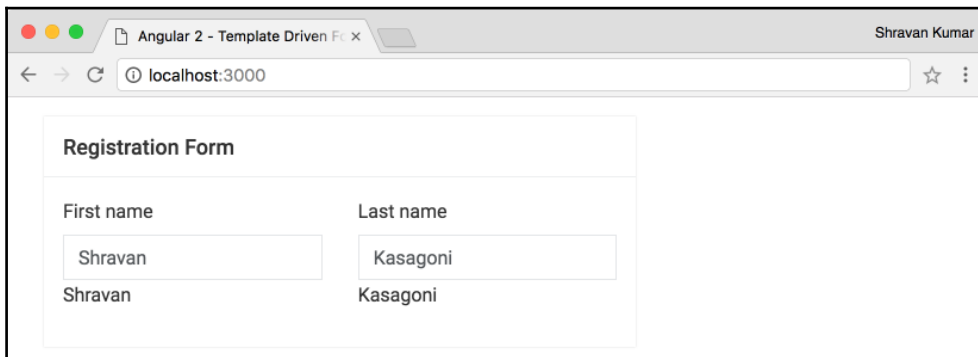

Using ngModel to bind a string value

If we need to bind an input control to an initial value, we can just specify `ngModel="<value>"`. This is a simple string binding. We are not using either property binding or event binding. The example for this is shown in the following code snippet:

```
<input type="text" class="form-control"
ngModel="Shravan" #firstNameRef="ngModel">
{{firstNameRef.value}}

<input type="text" class="form-control"
ngModel="Kasagoni" #lastNameRef="ngModel">
{{lastNameRef.value}}
```

Once we save the template, the browser will refresh with the following output:

Using ngModel to bind a component property

The code for `src/registration-form/registration-form.component.ts` is as follows:

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

interface User {
  firstName: string;
  lastName: string;
}

@Component({
  selector: 'registration-form',
  templateUrl: './registration-form.component.html'
})
export class RegistrationFormComponent {
  user: User = {
    firstName: 'Shravan',
    lastName: 'Kasagoni'
  }
}
```

We have a user object initialized inside the `RegistrationFormComponent` class. Let us use this user object to initialize the input controls on the template.

The code for `src/registration-form/registration-form.component.html` is as follows:

```
<input type="text" class="form-control"
                   ngModel="user.firstName"
#firstNameRef="ngModel">

{{firstNameRef.value}} <br/> {{user.firstName}}


<input type="text" class="form-control"
                   ngModel="user.lastName"
#lastNameRef="ngModel">
    {{lastNameRef.value}} <br/> {{user.lastName}}
```

If we look at the output in the browser, the input controls and their interpolation has displayed the `user.firstName` and the `user.lastName` strings directly instead of their values. Because we are using `ngModel="<value>"`, it is just a simple string binding, not property binding. Let us update our code to use property binding and set the initial values:

---

[102]

```
<input type="text" class="form-control"
                   [ngModel]="user.firstName"
#firstNameRef="ngModel">

<input type="text" class="form-control"
                   [ngModel]="user.lastName"
#lastNameRef="ngModel">
```

After updating the input controls with property binding, the component will start displaying the `user.firstName` and `user.lastName` property values. One interesting behavior we can observe, once we start changing the values in the input controls, their interpolation binding will update with whatever we type, but not the component properties. Because property binding is one-way binding, it will not update the property back with data changes in the input control.

To update the bound property with the updated value of the input control, we can use two-way data binding like this: `[(ngModel)]="user.firstName"`.

However, it is not recommended to use two-way data binding in template driven forms until it is necessary because we have to maintain the state in both the template and component which is unnecessary, and it might cause unknown problems.

How can we get the values of input controls back to the component?

We should use the form template reference variable and submit the value of it to the component. Let us remove all the code added to the template and component, put it back to the initial state of our example, and add a `<form>` tag to it:

```
<div class="box-body">
  <form novalidate>
    <div class="row">
      <div class="col-sm-6 form-group">
        <label>First name</label>
        <input type="text" class="form-control"
                          ngModel #firstNameRef="ngModel">
      </div>
      <div class="col-sm-6 form-group">
        <label>Last name</label>
        <input type="text" class="form-control"
                          ngModel #lastNameRef="ngModel">
      </div>
    </div>
    <button type="submit"
  class="btn btn-secondary">Submit</button>
  </form>
</div>
```

We added a `<form>` tag and a `submit` button inside of it; once the browser refreshes with the latest output, there are a lot of errors in the console:
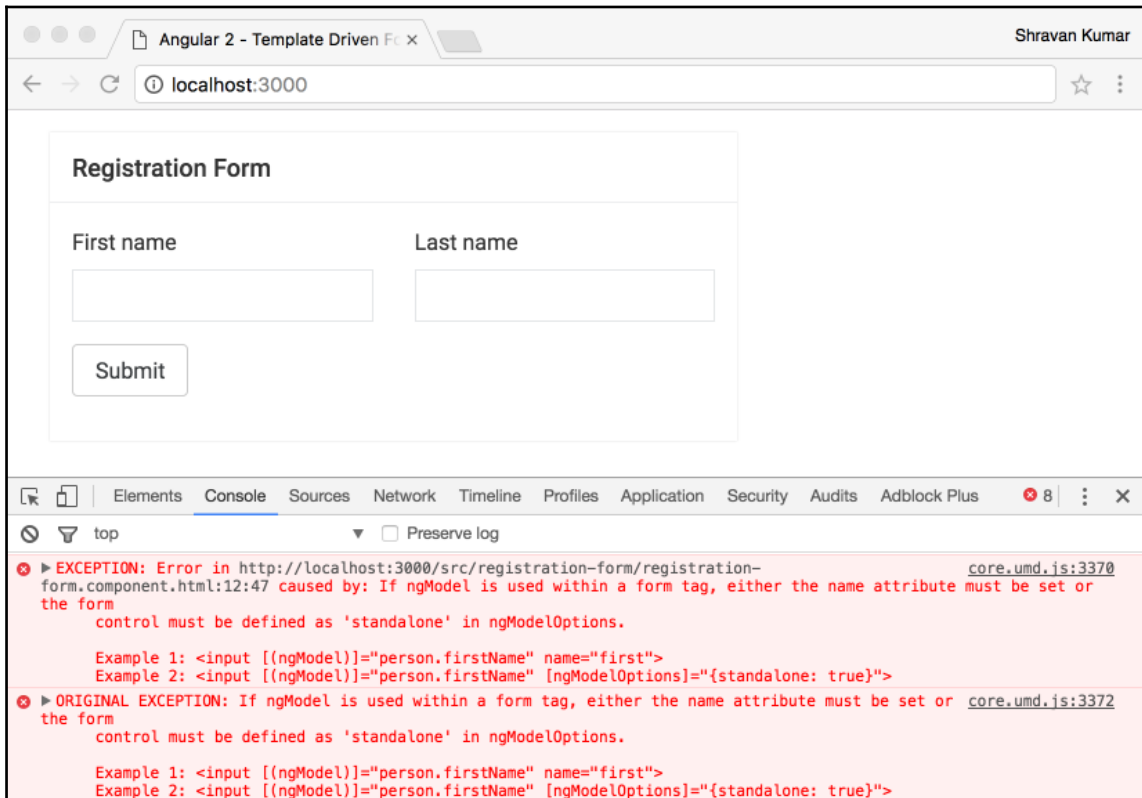
The error message pretty clearly says:

> *If ngModel is used within a form tag, either the name attribute must be set, or the form control must be defined as 'standalone' in ngModelOptions.*

When we use `ngModel` on the input control inside a `<form>` tag, we must declare a `name` attribute on it. The `ngModel` registers the input controls using their `name` attribute on the form. Let us add a `name` property to both our input controls:

```
<input type="text" class="form-control" name="firstName"
ngModel #firstNameRef="ngModel">

<input type="text" class="form-control" name="lastName"
ngModel #lastNameRef="ngModel">
```

---

[104]

Using the ngForm directive

The `ngForm` directive on the `form` tag represents the model (`FormGroup`) object, to access it we need to export to a template reference variable:

```
<form novalidate #formRef="ngForm"></form>
<pre>{{formRef | json}}</pre>
```

The code between `<form></form>` tags is removed for readability. We can use `#formRef` template reference variables to access the form model (`FormGroup`) objects, `#formRef` variable, and JSON pipe is used to display its entire structure of form model:
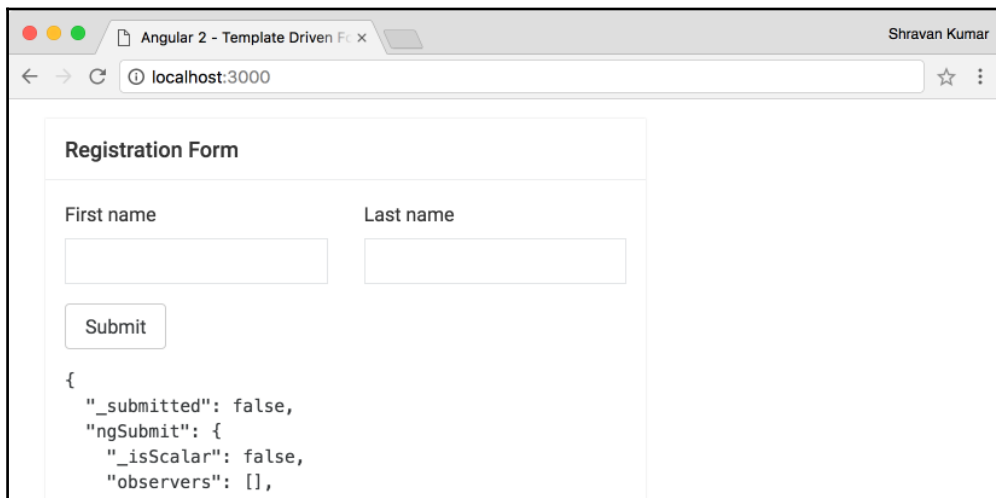
We do not need the entire structure; we simply need the value of the form. Let us use the value property on the `FormGroup` class because internally `ngForm` is a `FormGroup`:

```
<pre>{{formRef.value | json}}</pre>
```

Now we can view the JSON object displayed on the browser screen with the values of the input controls:
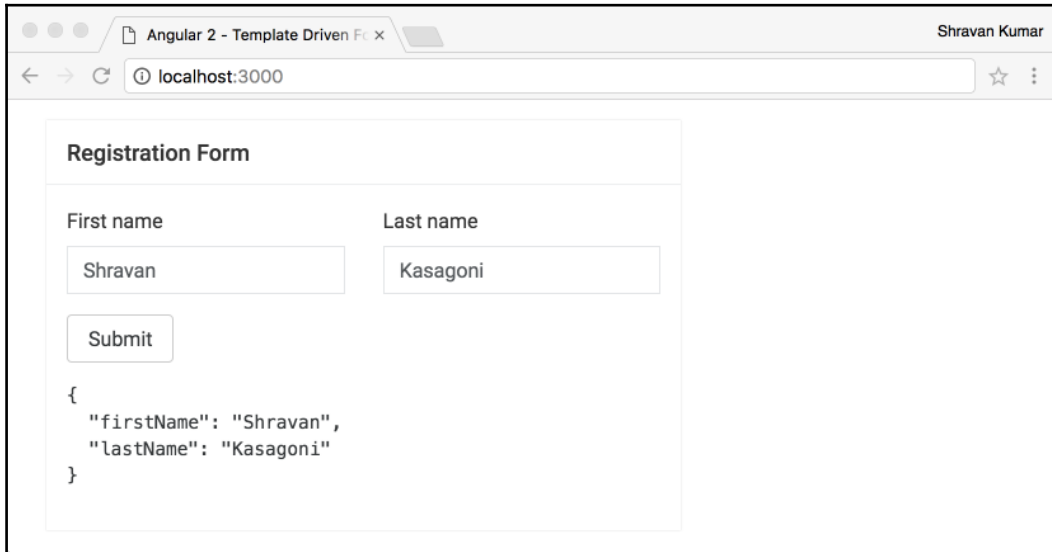
Interestingly, we did not add the `ngForm` directive on the `<form>` tag like we added the `ngModel` directive on the input controls. This is because whenever Angular encounters a `<form>` tag in the template, it will activate and attach the `ngForm` directive to the `<form>` tag implicitly, we do not need to add the `ngForm` directive on the `form` tag explicitly.

> If we do not want `ngForm` directive to automatically attach the `<form>` tag, we can disable this functionality by adding the `ngNoForm` directive as an attribute to the `<form>` tag.

Submitting a form using the ngSubmit method

To submit our form, we should use an `ngSubmit` event and attach it to a method in the component. Let us add a method to which we would display the value passed to it in the browser console:

```
export class RegistrationFormComponent {
  onSubmit(formValue) {
    console.log(formValue);
  }
}
```

Let us invoke the `onSubmit()` method whenever an `ngSubmit` event is triggered:

```
<form novalidate #formRef="ngForm"
                 (ngSubmit)="onSubmit(formRef.value)">
</form>
```

Now, once we type some data and click on the **Submit** button, it will trigger the `(ngSubmit)` event, which invokes the `onSubmit()` method on the component. To the `onSubmit()` method, we are passing our form value using `formRef.value` and displaying it in the browser console:
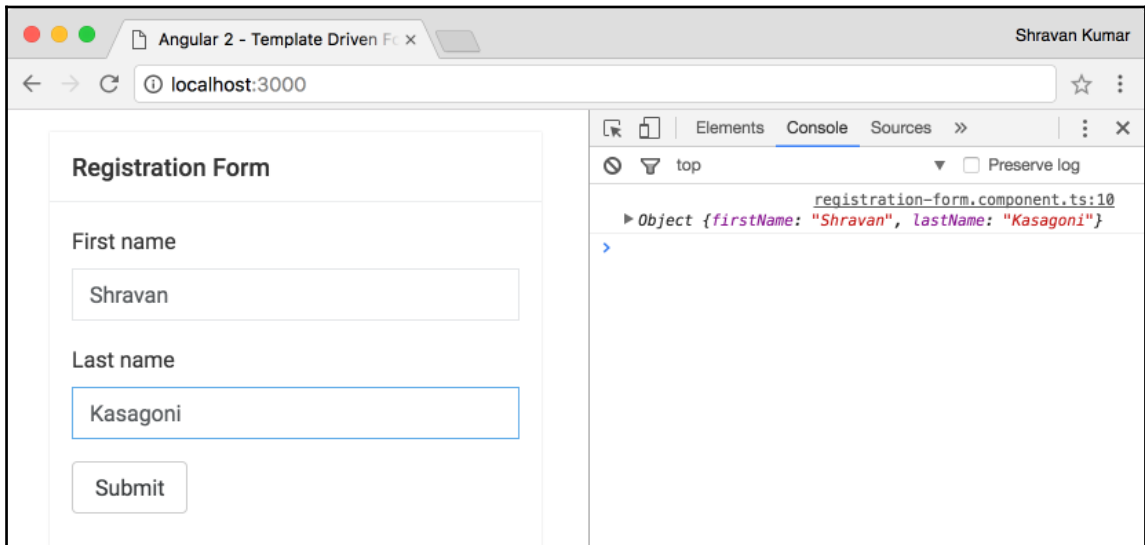
We are extending our example to use a couple of more fields and the most commonly-used controls (radio buttons, checkbox, and dropdown) in the forms.

Let us begin with adding `email`, `password`, and `confirmPassword` fields:

```
<div class="form-group">
  <label>Email</label>
  <input type="email" class="form-control" name="email"
                      ngModel #emailRef="ngModel">
</div>
<div class="row">
  <div class="col-sm-6 form-group">
    <label>Enter Password</label>
    <input type="password" class="form-control"
           name="password" ngModel #passwordRef="ngModel">
  </div>
```

```
    <div class="col-sm-6 form-group">
      <label>Confirm Password</label>
      <input type="password"
             class="form-control" name="confirmPassword"
             ngModel #confirmPassRef="ngModel">
    </div>
  </div>
  <div class="row">
    <div class="col-sm-6 form-group">
      <label>Street</label>
      <input type="text" class="form-control" name="street"
                         ngModel #streetRef="ngModel">
    </div>
    <div class="col-sm-6 form-group">
      <label>City</label>
      <input type="text" class="form-control" name="city"
                         ngModel #cityRef="ngModel">
    </div> </div>
```
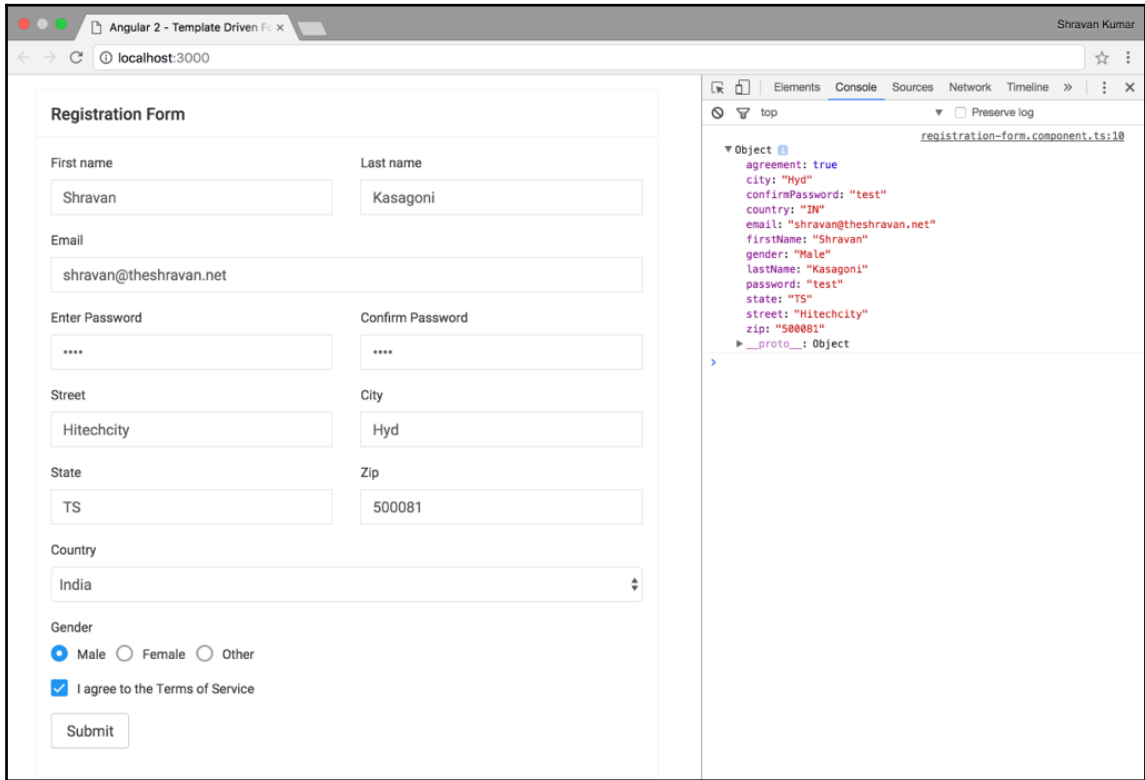
Now let us add street, city, state, zip, and country fields:

```
  <div class="row">
    <div class="col-sm-6 form-group">
      <label>Street</label>
      <input type="text" class="form-control" name="street"
                         ngModel #streetRef="ngModel">
    </div>
    <div class="col-sm-6 form-group">
      <label>City</label>
      <input type="text" class="form-control" name="city"
                         ngModel #cityRef="ngModel">
    </div>
  </div>
  <div class="row">
    <div class="col-sm-6 form-group">
      <label>State</label>
      <input type="text" class="form-control" name="state"
                         ngModel #stateRef="ngModel">
    </div>
    <div class="col-sm-6 form-group">
      <label>Zip</label>
      <input type="text" class="form-control" name="zip"
                         ngModel #zipRef="ngModel">
    </div>
  </div>
  <div class="form-group">
    <label>Country</label>
    <select class="form-control" name="country"
```

---

[108]

```
            ngModel="" #countryRef="ngModel">
    <option value="IN">India</option>
    <option value="US">United States of America</option>
  </select>
</div>
```

Let us add `gender` and `service` agreements fields:

```
<div class="row">
  <div class="col-sm-12 form-group">
    <label>Gender</label>
    <div>
      <label class="check-label">
        <input type="radio" name="gender" value="Male"
          ngModel #genderRef="ngModel"><i class="blue"></i>Male
      </label>
      <div class="spacer"></div>
      <label class="check-label">
        <input type="radio" name="gender" value="Female"
          ngModel #genderRef="ngModel"><i class="blue"></i>Female
      </label>
      <div class="spacer"></div>
      <label class="check-label">
        <input type="radio" name="gender" value="Other"
          ngModel #genderRef="ngModel"><i class="blue"></i>Other
      </label>
    </div>
  </div>
</div>
<div class="form-group">
  <label class="check-label">
    <input type="checkbox" name="agreement" value=""
      ngModel #agreementRef="ngModel"><i class="blue"></i>
      I agree to the Terms of Service
  </label>
</div>
```

Once we save the template, the browser refreshes with the latest out fill in the input and by clicking on the Submit button, we can view all the selected values as an object in displayed in the console. In real-world applications, we might send this data to the server using HTTP or we might perform some more operations on this data:
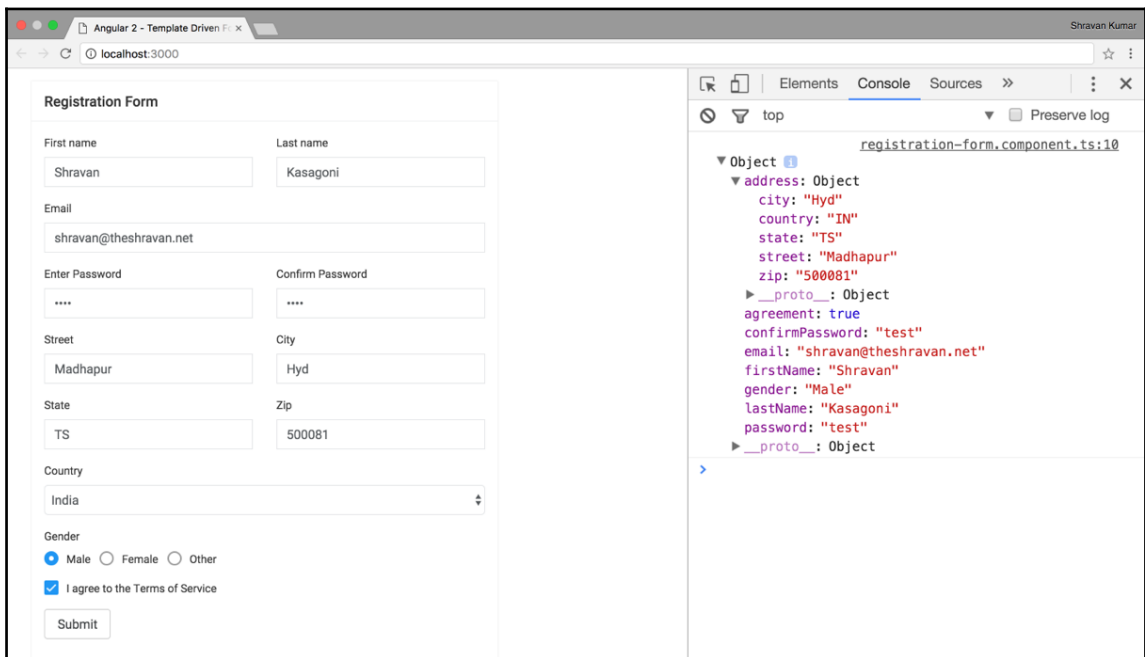

Using the ngModelGroup directive

As mentioned in the FormGroup section, sometimes it makes more sense to think of a series of form controls as a group and work with them. In our example, we have street, city, state, zip, and country and we are treating all these fields as individual controls, but all they represent is the address, so we treat all of them as the address group.

In the template driven to group the controls, we can use the `ngModelGroup` directive. Let us place all the `street`, `city`, `state`, `zip`, and `country` fields inside a `div` tag and attach it to the `ngModelGroup` directive:

```
<div ngModelGroup="address" #addressRef="ngModelGroup">
    <!-- street, city, state, zip and country fields code-->
</div>
```

Once again, we fill in the input and click on `Submit`. We can view all selected values as an object printed in the console and `street`, `city`, `state`, `zip`, and `country` properties are now under the `address` object, instead of the root:

The `ngModelGroup` value and state depend on all the controls inside of it; we can use its template reference variable, `#addressRef` to access the value, state, and other properties inside the template. Internally, the `ngModelGroup` directive is `FormGroup`.

Adding validations to the registration form

Before submitting any form, we need to validate whether the input entered by the user is correct or not. To apply the validation on the forms, Angular provides the following validation directives, we can also build our custom validators:

- `required`: Marks a control to have a non-empty value
- `minlength`: Applies minimum length validation
- `maxlength`: Applies maximum length validation
- `pattern`: Validates a control value to match a regex

Based on the input validity, validation directives change the state of the `ngModel`, `ngModelGroup`, and `ngForm`, we can access them using their template reference variables. Let us use the Angular validation directives to apply validation on our form controls. To demonstrate validation directives, we are going to use only the `firstName` input field. We can apply validation to the rest of the fields in a similar fashion.

Let us start with creating a project named `form-validations` from the previous example `forms`, change the name in the `package.json` file to `form-validations`:

```
<input type="text" class="form-control" name="firstName"
 ngModel #firstNameRef="ngModel" required>
<pre>{{firstNameRef?.errors | json}}</pre>
<pre>{{firstNameRef.valid}}</pre>
<pre>{{formRef.valid}}</pre>
```

We applied the `required` directive on our `firstName` field, the user must enter some input before submitting it:
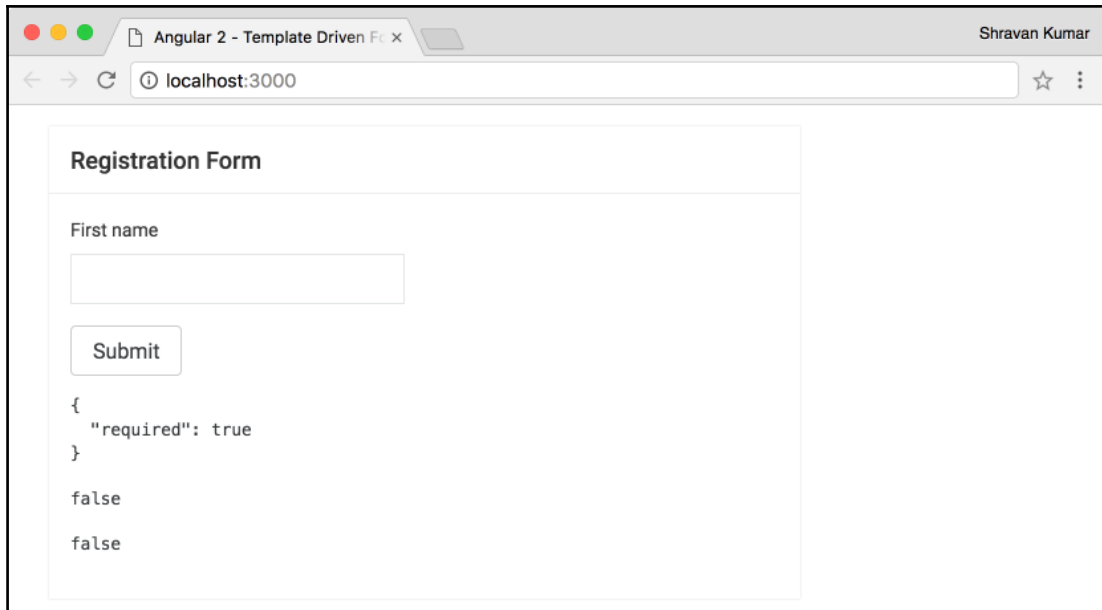
We are using the `errors` property on the form control object to get the list of errors. The `errors` property value is either an object or object of objects depending on the number of validations we apply on the form control. For each validation directive, `errors` property will have an object that contains information about it.

In our example, we applied `required` directive on the form control, and its `errors` property is `{ "required": true }`. It tells us that this field is required, we can use the information in `errors` property to display error messages and feedback to the user on the form.

The form control, `valid` property is returning `false` because its validation rules are not satisfied and the form `valid` property is also returning `false` because its state is calculated based on the control's state inside of it. Once all the controls inside the form become valid, its state also becomes valid.

Let us use the `errors` property to display an error message to the user:

```
<input type="text" class="form-control" name="firstName"
       ngModel #firstNameRef="ngModel" required
       [class.ctrl-error] = "firstNameRef.touched &&
                                firstNameRef?.errors?.required">
<div *ngIf="firstNameRef.touched &&
          firstNameRef?.errors?.required" class="error-message">
  The first name is required.
</div>
```

The output for the error message can be seen in the following screenshot:
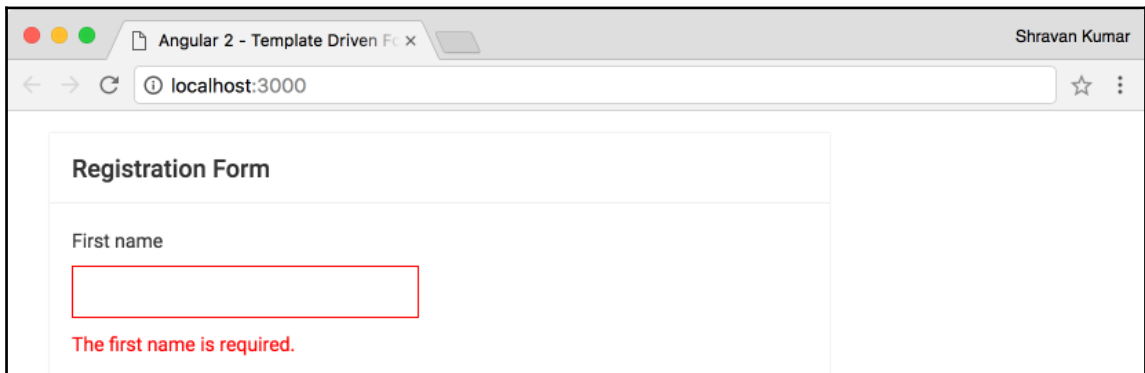
We are using `errors` and `touched` properties to display the error message as soon as the user leaves the **First name** textbox when entering the input. We are also using the same properties on the **First name** textbox to make the red color border using `class` binding.

We can apply as many validations we want on input controls. Let's use the `minlength` directive to implement a validation to force the user to enter a minimum of three characters into the **First name** textbox:

```
<input type="text" class="form-control" name="firstName"
       ngModel #firstNameRef="ngModel" required minlength="3">

<div class="error-message" *ngIf="firstNameRef.touched &&
                                firstNameRef?.errors?.required">
  The first name is required.
</div>

<div class="error-message" *ngIf="firstNameRef.touched &&
                            firstNameRef?.errors?.minlength">
You should enter minimum
{{firstNameRef?.errors?.minlength.requiredLength}}
characters into first name, but you entered only
```

```
{{firstNameRef?.errors?.minlength.actualLength}} characters.
</div>
```

If the user enters less than or equal to two characters in the **First name** textbox, the following error message will be displayed:
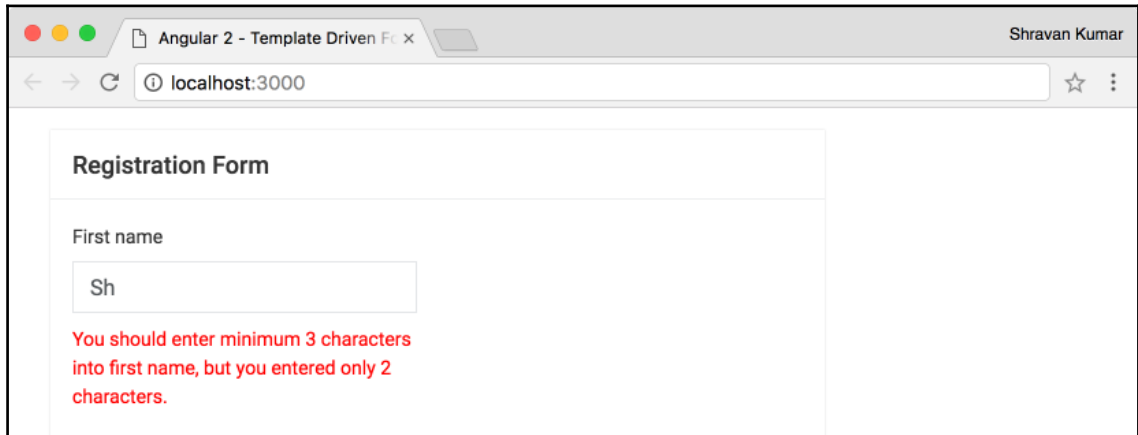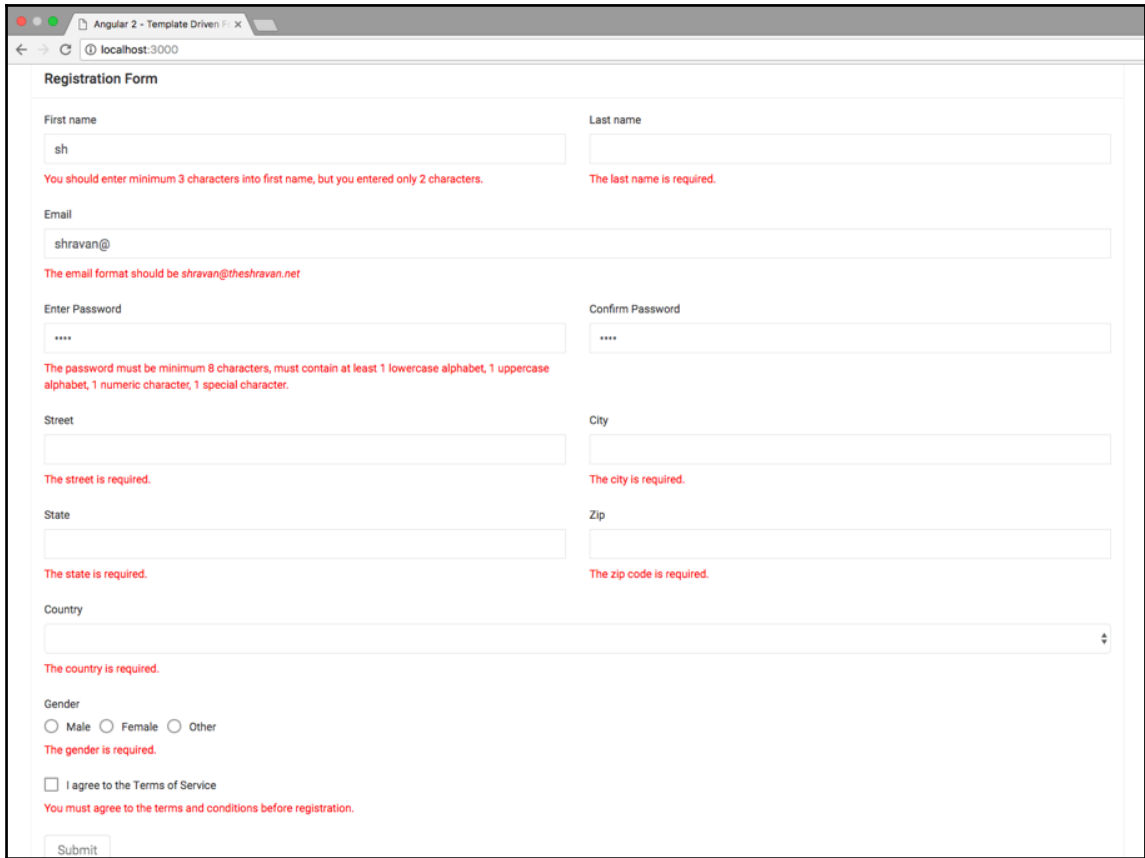

Let us do one last thing in our form. Even though there are errors, the user can submit it, which shouldn't be the case, so we will disable the Submit button if the form is invalid:

```
<button type="submit" class="btn btn-secondary"
                    [disabled]="formRef.invalid">Submit</button>
```

Here is an example of a fully-implemented form with all the validations. The sample code is under the `chapter5/form-validations` example:

Pros and cons of template driven forms

It is very easy to create large forms in a declarative fashion using template driven forms. However, the downside is all the logic, and validation rules are in HTML. It is difficult to unit test validation logic. We have to do end to end testing to verify the functionality using tools like protractor.

Reactive forms

Reactive forms (also known as model-driven forms) are the new approach introduced in Angular. In contrast to template driven forms in reactive forms, we will write all the form logic like creating controls, forms, and defining validation rules inside our component classes using the forms API instead of HTML.

In a reactive forms approach, we will directly use the classes `FormControl`, `FormGroup`, `FormArray`, and `FormBuilder` to create input controls, forms, and apply validation rules inside the `Component` class.

Let us create our previous example using the reactive forms approach.

Creating a registration form using reactive forms

To begin reactive forms in Angular, let us start by setting up a project named `reactive-forms` and using the following directory structure and files. Copy the previous code example:

```
reactive-forms
├── index.html
├── package.json
├── src
│   ├── app.component.ts
│   ├── app.module.ts
│   ├── main.ts
│   └── registration-reactive-form
│       ├── registration-reactive-form.component.html
│       └── registration-reactive-form.component.ts
├── styles.css
├── systemjs-angular-loader.js
├── systemjs.config.js
└── tsconfig.json
```

The code for `src/app.module.ts` is as follows:

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

import { AppComponent } from './app.component';
import { RegistrationReactiveFormComponent }
      from './registration- reactive-form
            /registration-reactive-form.component';

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

We must import and include `ReactiveFormsModule` to the imports array because all the form's API classes, `FormControl`, `FormGroup`, `FormArray`, `FormBuilder`, and `Validators` are available in that module.

The code for `src/app.component.ts` is as follows:

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

@Component({
  selector: 'forms-app',
  template: `<registration-reactive-form>
            </registration-reactive-form>`
})
export class AppComponent {
}
```

Using FormGroup, FormControl, and Validators

Let's use `FormGroup`, `FormControl`, and `Validators` classes to build our registration form inside the `Component` class.

The code for `src/registration-reactive-form/registration-reactive-form.component.ts` is as follows:

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

```
@Component({
  selector: 'registration-reactive-form',
  templateUrl: './registration-reactive-form.component.html'
})
export class RegistrationReactiveFormComponent implements OnInit {

  EMAIL_REGEX = "[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-
                  ]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*";

  registrationForm: FormGroup;

  ngOnInit() {
    this.registrationForm = new FormGroup({
      firstName: new FormControl('Shravan', Validators.required),
      lastName: new FormControl(''),
      email: new FormControl('', [Validators.required,
                          Validators.pattern(this.EMAIL_REGEX)])
    });
  }

  onSubmit(formValue) {
    console.log(formValue);
    console.log(this.registrationForm.value)
  }
}
```

The code is pretty much self-explanatory. At the start of the chapter, we discussed the `FormControl` and `FormGroup` classes.

In our component, we applied validation on the form controls using the `Validators` class which provides the same validation directives (`required`, `minlength`, `maxlength`, `pattern`) we discussed in template driven forms as a method. If we need to use multiple validation directives on the single form control, we can pass them in an array.

Using [formGroup], formControlName, and formGroupName

Now we need to bind `FormGroup` to the `<form>` tag using `[formGroup]` binding, and `FormControl` to the `<input>` tag using the `formControlName` directive:

```
<form novalidate [formGroup]="registrationForm"
                (ngSubmit)="onSubmit(registrationForm.value)">
  <input type="email" class="form-control"
                  formControlName="email">
  <div class="error-message"
```

```
        *ngIf="registrationForm.get('email').touched &&
                registrationForm.get('email').hasError('required')">
     The email is required.
   </div>
   <div class="error-message"
        *ngIf="registrationForm.get('email').touched &&
                registrationForm.get('email').hasError('pattern')">
     The email format should be <i>shravan@theshravan.net</i>
   </div>
   <button type="submit" class="btn btn-secondary"
           [disabled]="registrationForm.invalid">Submit</button>
 </form>
```

We are binding the `registrationForm` object (an instance of `FormGroup` class) to `[formGroup]`. E-mail input is attached to `email` property (an instance of a `FormControl` class) using the `formControlName` directive.

In reactive forms, we do not need the name property on input controls because they are created in the component and just bound to controls in the template. We also do not need a template reference variable; we can directly access the form controls using the `get()` method on the `FormGroup` class like `registrationForm.get('email')`, this will access all the methods and properties on the `FormControl` class.

We are accessing the validations on form controls using the `hasError()` method instead of the `errors` property, any approach works fine. The output would be pretty much the same.

To group the controls, we need to nest the form group inside another form group:

```
this.registrationForm = new FormGroup({
    firstName: new FormControl('Shravan', Validators.required),
    lastName: new FormControl(''),
    email: new FormControl('', [Validators.required,
                         Validators.pattern(this.EMAIL_REGEX)]),
    address: new FormGroup({
      street: new FormControl(''),
      country: new FormControl('', Validators.required)
    })
  });
```

We need to use the `formGroupName` directive to bind the group controls in HTML:

```
<div formGroupName="address">
  <div class="row">
    <div class="col-sm-6 form-group">
      <label>Street</label>
      <input type="text" class="form-control"
                          formControlName="street">
```

```
      </div>
    </div>
    <div class="form-group">
      <label>Country</label>
      <select class="form-control" formControlName="country">
        <option value="IN">India</option>
        <option value="UK">United Kingdom</option>
      </select>
      <div class="error-message" *ngIf="
          registrationForm.get('address').get('country')
                    .touched &&
      registrationForm.get('address').get('country')
                    .hasError('required')">
          The country is required.
      </div>
    </div>
  </div>
</div>
```

The remaining code in the template is removed for readability. The preceding code snippet should be inside `<form novalidate [formGroup]="registrationForm"></form>` tags.

Using FormBuilder

The `FormBuilder` class provides a simpler API to deal with control groups:

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

@Component({
  selector: 'registration-reactive-form',
  templateUrl: './registration-reactive-form.component.html'
})
export class RegistrationReactiveFormComponent implements OnInit {
  EMAIL_REGEX = "^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-9-
                ]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$";

registrationForm: FormGroup;

  constructor(public formBuilder: FormBuilder) { }

  ngOnInit() {
    this.registrationForm = this.formBuilder.group({
      firstName: ['Shravan', Validators.required],
      lastName: '',
```

```
        email: ['', [Validators.required,
                      Validators.pattern(this.EMAIL_REGEX)]],
        address: this.formBuilder.group({
            street: '',
            city: ['', Validators.required],
            state: ['', Validators.required],
            zip: '',
            country: ['', Validators.required]
        })
    });
  }
}
```

The `FormBuilder` class `group()` method returns the `FormGroup` object itself. Inside the group method, we are only passing the initial value and `Validators` for form control instead of creating a `FormControl` object manually every time. We do not need to make any changes in the template, it just works, this is a simplified API.

CustomValidators

In Angular, a validator is a simple function which accepts `AbstractControl` as an input parameter and returns an object literal where the key is *error code* and the value is `true` if it fails.

We want to use this `CustomValidators` in multiple components, let us create a class named `CustomValidators` and add our custom validation functions inside of it.

The code for `src/custom-validators.ts` is as follows:

```
import { AbstractControl } from '@angular/forms';

export class CustomValidators {

  static passwordStrength (control: AbstractControl) {

    if (CustomValidators.isEmptyValue(control.value)) {
      return null;
    }

    if (!control.value.match(/^(?=
            .*[0-9])(?=.*[!@#\$%\^&\*])(?=.*[a-z])
            (?=.*[A-Z])[a-zA-Z0-9!@#\$%\^&\*]{8,}$/)) {
      return {'weakPassword': true};
    }
    retrun null;
```

```
  }

  static isEmptyValue (value) {
    return value == null ||
          typeof value === 'string' && value.length === 0;
  }
}
```

We created a static method, `passwordStrength()` which accepts control as a parameter and compares its value against a regular expression to check the strength of the password and returns an error object if the control value does not meet the regular expression criteria:

```
import { CustomValidators } from '../CustomValidators';

ngOnInit () {
this.registrationForm = this.formBuilder.group({
  password: ['', [Validators.required,
                  CustomValidators.passwordStrength]]
});
}
```

Inside the template, we should have the same property (`weakPassword`) in the error object returned by `CustomValidators` using the `hasError('weakPassword')` method:

```
<input type="password" class="form-control"
                      formControlName="password" >
<div class="error-message"
    *ngIf="registrationForm.get('password').touched &&
          registrationForm.get('password').hasError('required')">
The password is required.
</div>
<div class="error-message"
    *ngIf="registrationForm.get('password').touched &&
        registrationForm.get('password').hasError('weakPassword')">
The password must be minimum 8 characters, must contain at
least 1 lowercase alphabet, 1 uppercase alphabet, 1 numeric
character, 1 special character.
</div>
```

We learned how to apply the custom validation on a single control, but sometimes our logic depends on multiple control values in that we should use the validators at the group level. Let us create one more validator which compares both password and confirm password values and returns an error object if both passwords do not match:

```
import { AbstractControl } from '@angular/forms';

export class CustomValidators {

  static passwordMatcher(control: AbstractControl) {

    const password = control.get('password').value;
    const confirmPassword = control.get('confirmPassword').value;

    if (CustomValidators.isEmptyValue(password) ||
        CustomValidators.isEmptyValue(confirmPassword)) {
      return null;
    }

    return password === confirmPassword ? null
                                        : { 'mismatch': true };
  }

  static isEmptyValue(value) {
    return value == null ||
           typeof value === 'string' && value.length === 0;
  }
}
```

We can use the `passwordMatcher()` validator at the form group level in the component and its error object in the template:

```
ngOnInit () {
  this.registrationForm = this.formBuilder.group({
    password: ['', [Validators.required,
                    CustomValidators.passwordStrength]],
    confirmPassword: ['', Validators.required],
  }, {validator: CustomValidators.passwordMatcher});
}
```

In the template:

```
<form novalidate [formGroup]="registrationForm">
  <!--Remain code is removed for readability-->
  <input type="password" class="form-control"
        formControlName="confirmPassword">
  <div class="error-message" *ngIf="
    registrationForm.get('confirmPassword').touched &&
    registrationForm.get('confirmPassword').hasError('required')">
The confirm password is required.
  </div>
  <div class="error-message" *ngIf="
registrationForm.get('confirmPassword').touched &&
registrationForm.hasError('mismatch')">
    The confirm password should match password.
  </div>
</form>
```

Pros and cons of reactive forms

As mentioned earlier, the reactive forms approach is new in Angular. It is very easy to define complex forms in code instead. As we are writing all the validation logic in the components, unit testing our form's logic is quite easy without any dependency on DOM, by simply instantiating the classes.

Summary

We started this chapter with a discussion on why developing forms is harder, and then we discussed different kinds of approaches in Angular that makes the development easier. We learned how to build the template driven forms and reactive forms and pros and cons of both methods. We also learned how to use built-in validations and how to write CustomValidators.

By the end of this chapter, the reader should have a good understanding of how to build forms using different APIs in Angular.

6

Building a Book Store Application

In this chapter, we will learn how to implement some real-world application scenarios by developing a Book Store application. After going through this chapter, the reader will understand the following concepts:

- Communicating with the REST service using an HTTP client
- Navigating between the components using routing
- Animations
- NgRX
- Feature modules

Book Store application

We are going to learn how to develop a Book Store application using various Angular concepts. The Book Store application consists of different components related to features provided by a real book store, where we can view the available list of books and each book's information, add new books, and delete old books. Before we start developing our Book Store application, you are going to learn how to use an HTTP client in Angular.

HTTP

Any Angular application that needs to communicate with the backend using REST services needs an HTTP client. Angular comes with its own HTTP library; it is available in the `@angular/http` npm package.

Before we start learning about the HTTP library, we need an application where we can use it. We are going to use Angular CLI to create our project; before getting started, make sure that you have Angular CLI installed on your machine.

Run the following command to install Angular CLI:

```
$ npm install -g @angular/cli@latest
```

Run the following command to create an Angular project using CLI:

```
$ ng new http-client-basics
```

The preceding command will create the Angular application with all the required libraries and tooling. Now navigate to our project folder and start the application using the following commands:

```
$ cd http-client-basics
$ npm start
```

Now the project is running at: `http://localhost:4200`. Navigate to the URL in the browser and we can view the output.

> Learn more about Angular CLI at: `https://cli.angular.io`.

We need a little more setup before we start writing the code to use HTTP client. Our HTTP client needs to connect to a REST service to get the data; for this example's purpose, we are going to use the JSON server npm package to create a fake REST API. We can replace this with any real REST API. Follow these steps to use the JSON server:

1. Install the JSON server npm package in our application root directory.

   ```
   $ npm install json-server --save-dev
   ```

2. Add the following command to the `scripts` section in the `package.json` file to run the JSON server.

   ```
   "json-server": "json-server --watch db.json --port 4567"
   ```

Copy the db.json file to our application root directory. The file contains book information, which we used in Chapter 3, *Components, Services, and Dependency Injection*, (you can copy this file from source code under chapter6/http-client-basics). Now run the following command to invoke the JSON server:
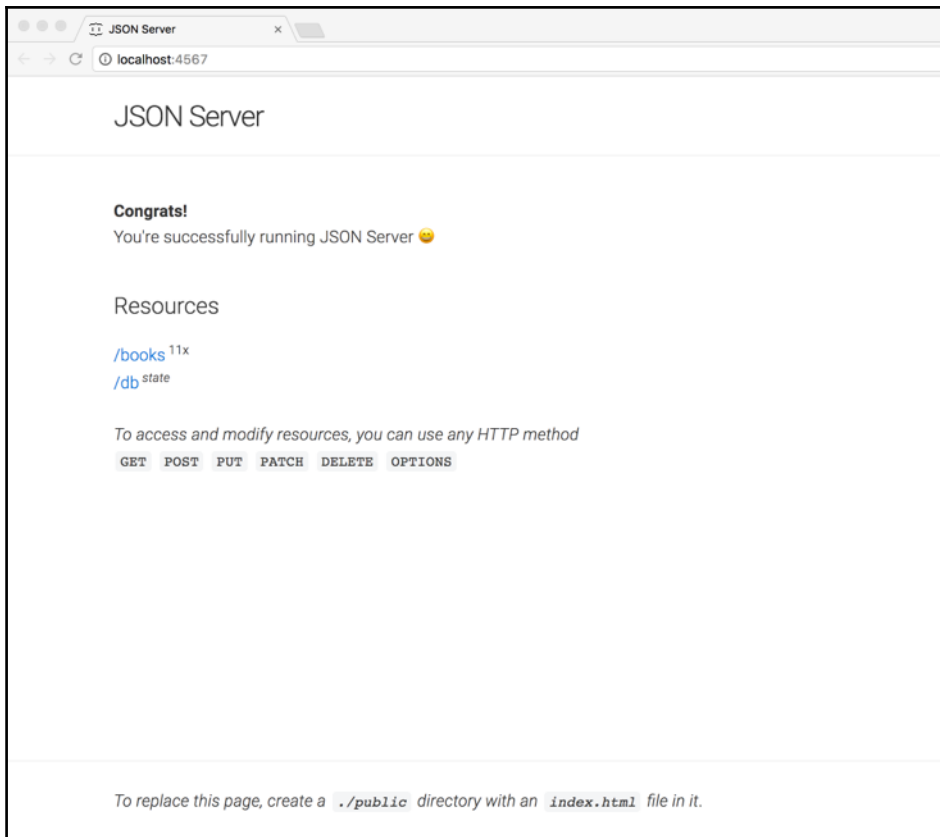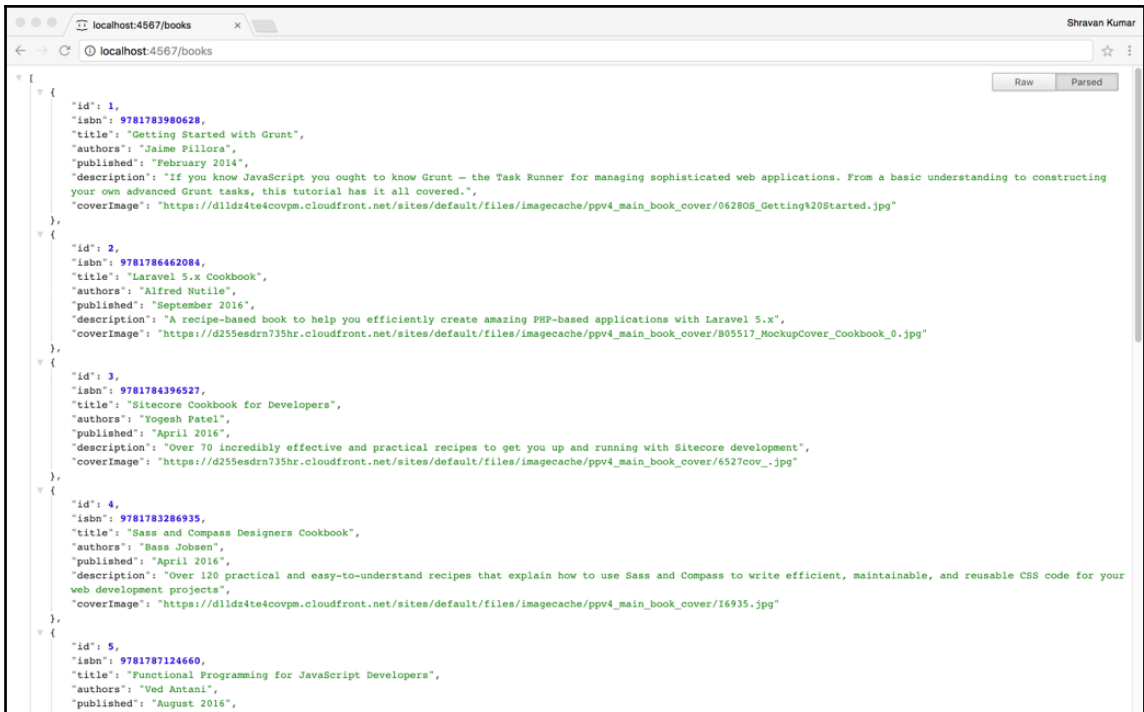
```
$ npm run json-server
```

The preceding command will start our API at the URL http://localhost:4567. We can navigate to this URL in the browser to check the functionality.

> Learn more about the JSON server
> at: https://github.com/typicode/json-server.

We have our API ready; let us write some code to communicate with the API. As mentioned in the beginning, we need the `@angular/http` package to work with the HTTP client; Angular CLI has already downloaded the npm package when we created the project.

Import `HttpModule` in our application module (`src/app/app.module.ts`).

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

Add the `HttpModule` to the `imports` array in the `@NgModule()` decorator:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Add the `Book` class under the app folder, which represents the book object structure.

The code for `src/app/book.ts` is as follows:

```
export class Book {
  id: number;
  isbn: number;
  title: string;
  authors: string;
  published: string;
  description: string;
  coverImage: string;
}
```

Let us add some code to our application component template (`app.component.html`).

```
<button (click)="getBooksData()">Get Books Data</button>
<pre>{{booksList | json}}</pre>
```

In the earlier template, we are calling the `getBooksData()` method on the `Component` class whenever the button is clicked, and we also display the `booksList` array in JSON format using the `json` pipe. We should define the `getBooksData()` method, and the `booksList` array on the `Component` class (`app.component.ts`):

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

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

  booksList: Book[] = [];

  getBooksData() {
    console.log(this.booksList);
  }
}
```

Making GET requests

The HTTP client is available as `Http` service in the `@angular/http` package; import it in our component and inject it via dependency injection into a constructor:

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


constructor(private http: Http) { }
```

We have our HTTP client; now invoke the API to get the data whenever the user clicks on the **Get Books Data** button:

```
getBooksData() {
    this.http.get('http://localhost:4567/books')
        .subscribe(res => this.booksList =
                            res.json() as Book[]);
}
```

We are calling our API using the GET method. Angular `Http` service is an `Observable`, and we need to subscribe to it to receive the response. Once the preceding code is added to the `getBooksData()` method, if we click on the button, we will receive all of the books' information in JSON format from the API.
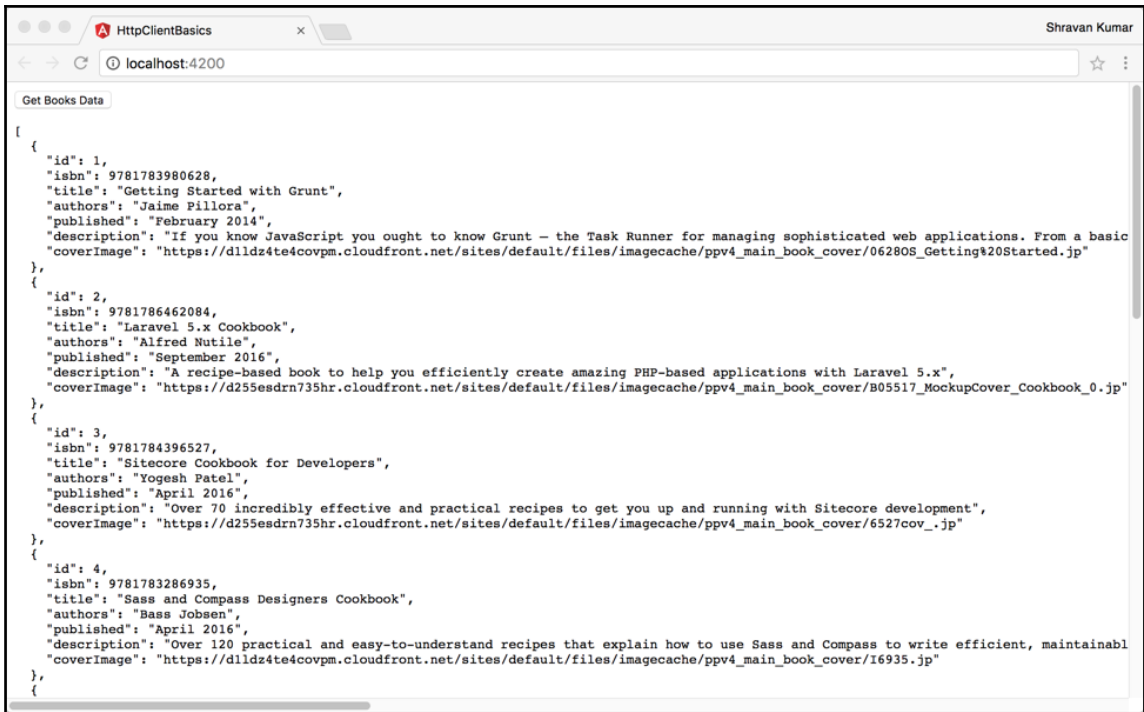
We are just displaying all our responses on the template, which is not very useful. Let us change it into a presentable format for the user.

The code for `src/app/app.component.html` is as follows:

```html
<div>
  <div class="left-container">
    <ul>
      <li *ngFor="let book of booksList"
                  (click)="getBookInfo(book.id)">
        {{book.title}}
      </li>
    </ul>
  </div>
  <div *ngIf='book' class="right-container">
    <p>{{book.isbn}}</p>
    <p>{{book.title}}</p>
    <p>{{book.authors}}</p>
    <p>{{book.published}}</p>
    <p>{{book.description}}</p>
    <p>
      <img [src]="book.coverImage" [alt]="book.isbn" />
```

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

We update our template to display the list of books on the left side of the page. Whenever the user clicks on the book name, we are calling our API using the HTTP client to get specific book information, and the response is displayed on the right side of the page.

We have all the information-related books. Still, we are calling the API to get the specific book information using the ID for this example's purpose only. A real-world API should return only the required information, and we update our component to call the API to get the specific book's information:

The code for `src/app/app.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Http } from '@angular/http';
import { Book } from './book';

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

  booksList: Book[] = [];
  book: Book;
  baseUrl: string = 'http://localhost:4567';

  constructor(private http: Http) { }

  ngOnInit() {
    this.getBooksData();
  }

  getBooksData() {
    const url = `${this.baseUrl}/books`;
    this.http.get(url)
        .subscribe(res => this.booksList =
                          res.json() as Book[]);
  }

  getBookInfo(id: number) {
    const url = `${this.baseUrl}/books/${id}`;
    this.http.get(url)
        .subscribe(res => this.book = res.json() as Book);
```

```
    }
  }
```

Here is the output:
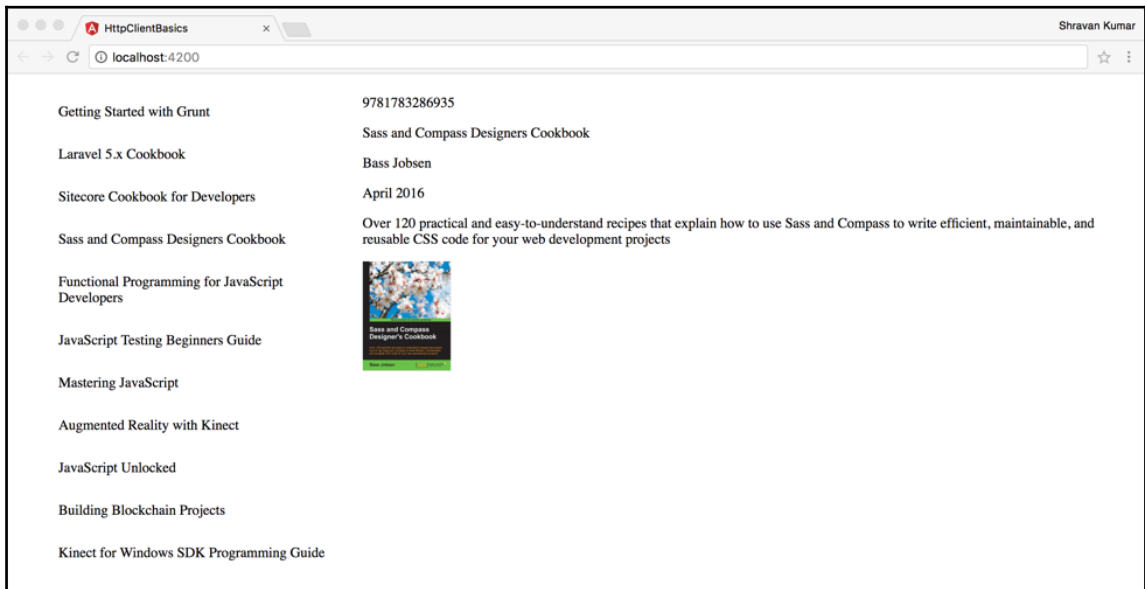
Let us refactor our code before moving on to the next section. In our example, `AppComponent` is directly communicating with API using the `Http` service. This is the responsibility of an Angular service. Move all the logic related to API communication to a *Book Store* service:

The code for `src/app/book-store.service.ts` is as follows:

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import { Book } from './book';

@Injectable()
export class BookStoreService {

  baseUrl: string = 'http://localhost:4567';

  constructor(private http: Http) { }

  getBooksList(): Observable<Book[]> {
```

```
    const url = `${this.baseUrl}/books`;
    return this.http.get(url)
               .map(response => response.json() as Book[]);
  }

  getBook(id: number): Observable<Book> {
    const url = `${this.baseUrl}/books/${id}`;
    return this.http.get(url)
               .map(response => response.json() as Book);
  }
}
```

We need to import and add the `BookStoreService` to the `providers` array in the `AppModule` before we can start using it:

The code for `src/app/app.module.ts` is as follows:

```
import { BookStoreService } from './book-store.service';

@NgModule({
    ...
providers: [BookStoreService],
...
})
export class AppModule { }
```

Here is the refactored `AppComponent` using the `BookStoreService` to get the data from the API:

The code for `src/app/app.component.ts` is as follows:

```
import { BookStoreService } from './book-store.service';

export class AppComponent implements OnInit {

  booksList: Book[] = [];
  book: Book;

  constructor(private bookStoreService: BookStoreService) { }

  ngOnInit() {
    this.getBooksData();
  }

  getBooksData() {
    this.bookStoreService.getBooksList()
        .subscribe(books => this.booksList = books);
```

```
    }

    getBookInfo(id: number) {
      this.bookStoreService.getBook(id)
        .subscribe(book => this.book = book);
    }
  }
```

In the next section, you are going to learn about routing, and we will be using a sample Book Store application. At the end of `Chapter 3`, *Components, Services, and Dependency Injection*, we created a master-detail application. The sample application is recreated using *Material Design Lite* for styling, HTTP client to get the data, and it is available under the `Chapter6/start` folder in the provided source code.

We can use the application under the `Chapter6/start` folder as a starting point to follow along remaining of this chapter. Let us create the folder named `book-store` and copy all the files and folders from the `Chapter6/start` directory.

Run the following commands in the root of the `book-store` folder before we get started with routing:

```
$ npm install
$ npm run json-server
$ npm start
```

Navigate to `http://localhost:4200` in the browser to view the Book Store application.

> Learn more about Material Design Lite at: `https://getmdl.io`.
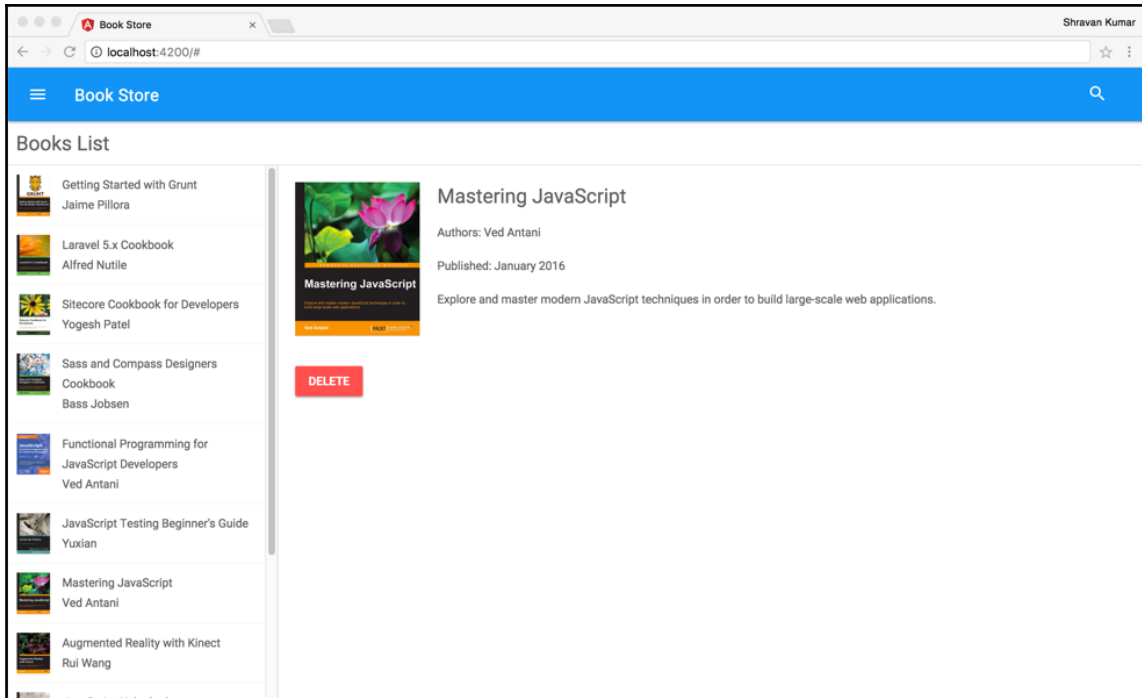
Routing

In the previous chapters, you learned different concepts in Angular to build applications. All our examples contain a maximum of two components. Any real-world application contains many components; we should be able to navigate between the different pages/components in the application, pass the data from one component to another component, and update multiple components in the same component tree. Angular comes with its own router, which is available in the `@angular/router` npm package.

Defining routes

To get started with the router, we need to follow these steps:

- Set the base `href`
- Import the `RouterModule` into `AppModule`
- Define the `routes` array using the `Routes` object
- Add the routes to the import array using the `RouterModule.forRoot()` method
- `index.html`

The browser uses the base `href` value to prefix relative URLs when referencing CSS, JS, and image files. Here is an example of `href`:

```
<head>
<base href="/">
</head>
```

The code for `src/app/app.module.ts` is as follows:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { RouterModule, Routes } from '@angular/router';

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

import {
  BooksListComponent,
  BookDetailsComponent,
  NewBookComponent,
  BookStoreService
} from './books';

import { Safe } from './safe';

const routes: Routes = [
  {path: '', redirectTo: 'books', pathMatch: 'full'},
  {path: 'books', component: BooksListComponent},
  {path: 'books/new', component: NewBookComponent},
  {path: 'books/:id', component: BookDetailsComponent},
  {path: 'about', component: AboutComponent},
];

@NgModule({
```

```
  declarations: [
    AppComponent, AboutComponent, BooksListComponent,
    BookDetailsComponent, NewBookComponent, Safe
  ],
  imports: [
    BrowserModule, ReactiveFormsModule,
    HttpModule, RouterModule.forRoot(routes)
  ],
  providers: [BookStoreService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Earlier in `AppModule`, we defined our `routes` array using the `Routes` object. Each route specifies the current router state. The `Routes` object has many properties, and we are using some of them to define routes for the Book Store application. The explanation for the different types of routes we specified are as follows:

```
{path: '', redirectTo: 'books', pathMatch: 'full'}
```

If we look at our first route, the path property is empty; we specified the `redirectTo` property. Whenever we launch the application start with /, it will redirect to the `books` path and display its corresponding component:

```
{path: 'books', component: BooksListComponent}
```

Our second path is very simple; whenever the path is `books`, it will show the `BooksListComponent`:

```
 {path: 'books/:id', component: BookDetailsComponent}
```

Our fourth path is bit different; it has two segments. The first segment is `books`, and it is a simple string match. The second segment is `:id` and specifies the parameter for the route.

Earlier, in our routes for different paths, we specified `BooksListComponent`, `NewBookComponent`, and `AboutComponent`, but we have not created these components in our application. We also need a placeholder in our application to display these components.

RouterOutlet Directive

The `RouterOutlet` directive acts as a placeholder where Angular can dynamically show the components based on the current router state.

In our application, till now we are displaying everything in `AppComponent`. We will be using `AppComponent` as a placeholder to show the common elements and other components based on the routes. We have a header and left-hand side menu, and both are common across the application. We will keep them in as it is in `AppComponent`, and the remaining space will display the other components using the `RouterOutlet` directive.

The code for `src/app/app.component.ts` is as follows:

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

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

We removed all the logic from `AppComponent` because it is going to act as a placeholder. If we look at the template here, we removed all the code under the `<main></main>` tag and added `<router-outlet></router-outlet>`, under the `<main>` tag to display the other components:

The code for `src/app/app.component.html` is as follows:

```
<main class="mdl-layout__content page-content">
    <router-outlet></router-outlet>
 </main>
```

Earlier remaining code is removed for more readability; we can find the complete code from the provided source code.

Let us create the `BooksListComponent` to display the list of books:

The code for `src/app/books/books-list/books-list.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../book';
import { BookStoreService } from '../book-store.service';

@Component({
  selector: 'books-list',
  templateUrl: './books-list.component.html',
  styleUrls: ['./books-list.component.scss']
})
export class BooksListComponent implements OnInit {
```

```
    booksList: Book[];

    constructor(private storeService: BookStoreService) {
    }

    ngOnInit() {
      this.getBooksList();
    }

    getBooksList() {
      this.storeService.getBooks()
        .subscribe(books => this.booksList = books);
    }
  }
```

The preceding component fetches the list of books from `BookStoreService`. As soon as the application is loaded, the router will redirect to the `BooksListComponent` and display the list of books. When we click **VIEW BOOK** link, we will navigate to `BookDetailsComponent` to display a particular book's information:
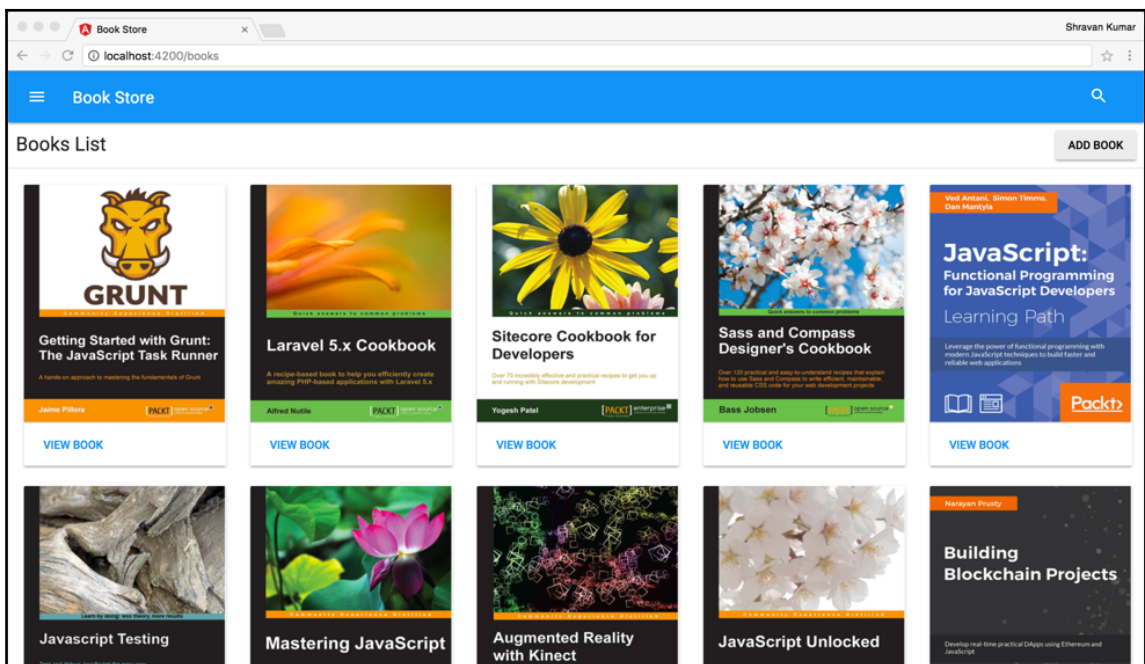
Named RouterOutlet

We can use named outlets to load multiple components side by side instead of nesting them. Named outlet is created by specifying the name attribute on the `RouterOutlet` directive. We can have one primary outlet (unnamed outlet), as many named outlets:

```
<router-outlet></router-outlet>
<router-outlet name="secondary"></router-outlet>
```

We specify the targeted outlet while defining the route itself or while navigating to the route imperatively or declaratively.

Navigation

The Angular router provides two ways to navigate from one component to another. The declarative way using the `RouterLink` directive is as follows:

```
<a routerLink="/books/new">Add Book</a>
```

We can specify the path for the `routerLink` directive as a string, and we can also generate the path dynamically by binding it to an array using the property binding:

```
<a [routerLink]="['/books', book.id]">View Book</a>
```

Earlier, two code snippets were used in the `BooksListComponent` template for navigation. We can also navigate from one component to other components imperatively using `navigate()` and `navigateByUrl()` methods in the `Router` object; we will be using them in the next component (`BookDetailsComponent`) to navigate back to `BooksListComponent`.

Route params

We can pass values when navigating from one component to the other. In our example, we are passing the `id` value from `BooksListComponent` to `BookDetailsComponent`. We can access route parameters using the `ActivatedRoute` object `Params` property:

The code for `src/app/books/book-details/book-details.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router }
                              from '@angular/router';
import { Location } from '@angular/common';
import 'rxjs/add/operator/switchMap';
import { BookStoreService } from '../book-store.service';
import { Book } from '../book';

@Component({
  selector: 'book-details',
  templateUrl: './book-details.component.html',
  styleUrls: ['./book-details.component.scss']
})
export class BookDetailsComponent implements OnInit {

  book: Book;

  constructor(private route: ActivatedRoute,
              private router: Router,
              private location: Location,
              private storeService: BookStoreService) {
  }

  ngOnInit(): void {
    this.route.params.switchMap((params: Params) =>
this.storeService.getBook(+params['id']))
                .subscribe(book => this.book = book);
  }

  deleteBook(id: number) {
    this.storeService.deleteBook(id)
        .subscribe(res => this.router.navigate(['/books']));
  }

  goBack() {
    this.location.back();
  }
}
```

As soon as the component is initialized, we are use `ActivatedRoute` to access the route parameters using the `Params` property, which is an Observable. We are use the `switchMap()` operator on the `Params` Observable to receive the latest parameters, and then we are invoke the `BookStoreService` and pass the `id` as a parameter to the `getBook()` method using `+params['id']`.
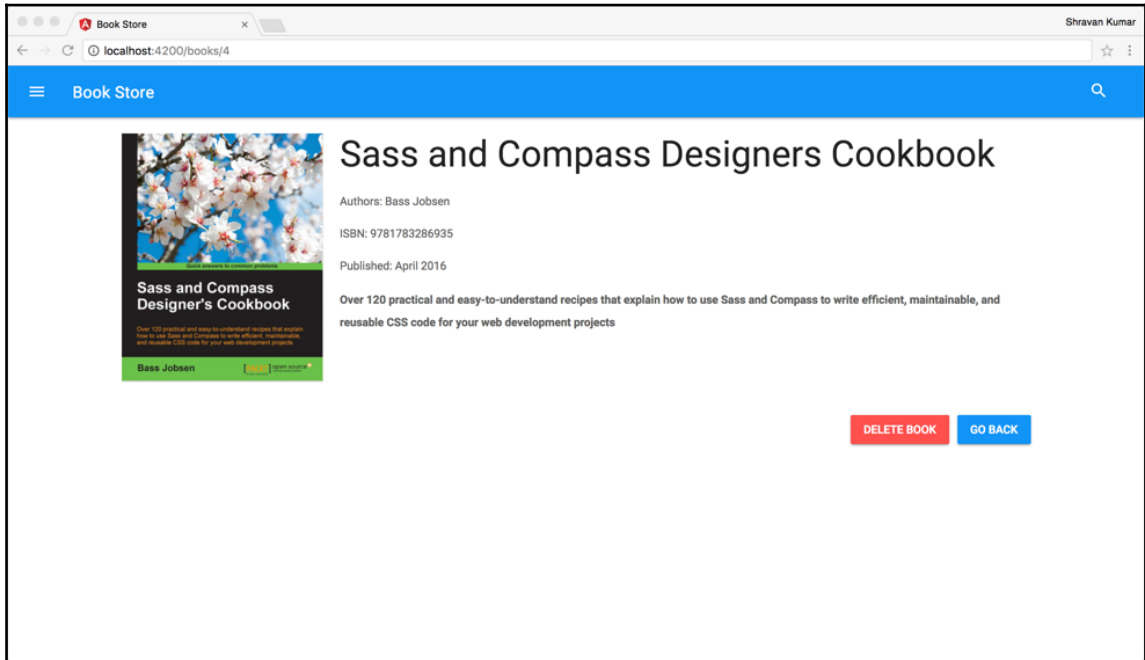
When we are on the component, if the route parameters change, the router does not need to re-activate the entire component because `Params` is an Observable, and it will receive the new values and emits them. The `switchMap()` operator always subscribes to the latest Observable, and it will always use the most recent values and executes the code. In our case, it gets the data from service using the `id` parameter.

We have a `deleteBook()` method in the component which invokes the `deleteBook()` method in `BookStoreService`. As soon we receive the response from the service, we use the `Router` object `navigate()` method to go back to `BooksListComponent`. We are also using the `Location` object `back()` method to go back to the previous route; `Location` object uses the browser history to navigate backward and forward.

Here is the implementation of `NewBookComponent` using reactive forms:

The code for `src/app/books/new-book/new-book.component.ts` is as follows:

```typescript
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from
                                    '@angular/forms';
import { Router } from '@angular/router';
import { Location } from '@angular/common';
import { Book } from '../book';
import { BookStoreService } from '../book-store.service';

@Component({
  selector: 'new-book',
  templateUrl: './new-book.component.html',
  styleUrls: ['./new-book.component.scss']
})
export class NewBookComponent implements OnInit {

  newBookForm: FormGroup;

  constructor(private formBuilder: FormBuilder,
              private router: Router,
              private location: Location,
              private storeService: BookStoreService) {
  }

  ngOnInit() {
    this.newBookForm = this.formBuilder.group({
      isbn: ['', Validators.required],
      title: ['', Validators.required],
      authors: ['', Validators.required],
      published: ['', Validators.required],
      description: ['', Validators.required],
      coverImage: ['', Validators.required]
    });
  }
  saveBook() {
    if (this.newBookForm.valid) {
      var book = this.newBookForm.value as Book;
      this.storeService.addBook(book)
        .subscribe(res => this.router.navigate(['/books']));
    }
  }
}
```

The code for `src/app/books/new-book/new-book.component.html` is as follows:

```html
<section class="new-book-container">
  <h4>Add Book</h4>
  <form novalidate [formGroup]="newBookForm"
(ngSubmit)="saveBook()">
    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label">
      <input class="mdl-textfield__input" type="text" id="isbn"
formControlName="isbn">
      <label class="mdl-textfield__label"
for="isbn">ISBN</label>
    </div>
    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label">
      <input class="mdl-textfield__input" type="text"
id="title" formControlName="title">
      <label class="mdl-textfield__label" for="title">
Book Title</label>
    </div>
    <br/>

    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label">
      <input class="mdl-textfield__input" type="text"
id="authors" formControlName="authors">
      <label class="mdl-textfield__label"
for="authors">Authors</label>
    </div>
    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label ">
      <input class="mdl-textfield__input" type="text"
id="published" formControlName="published">
      <label class="mdl-textfield__label"
for="published">Published</label>
    </div>
    <br/>

    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label floating-label-full">
      <input class="mdl-textfield__input" type="text"
id="description" formControlName="description">
      <label class="mdl-textfield__label"
for="description">Description</label>
    </div>
    <br/>
    <div class="mdl-textfield mdl-js-textfield mdl-textfield-
floating-label floating-label-full">
```

```
        <input class="mdl-textfield__input" type="text"
id="coverImage" id="coverImage"
formControlName="coverImage">
        <label class="mdl-textfield__label"
for="coverImage">Cover Image</label>
    </div>
    <br/>

    <button type="submit" class="mdl-button mdl-js-button mdl-
button--raised mdl-js-ripple-effect mdl-button--colored"
[disabled]="newBookForm.invalid">
        Save
    </button>

    <button type="submit" class="mdl-button mdl-js-button mdl-
button--raised mdl-js-ripple-effect mdl-button--accent"
(click)="this.location.back()">
        Cancel
    </button>
  </form>
</section>
```

Here is the output:
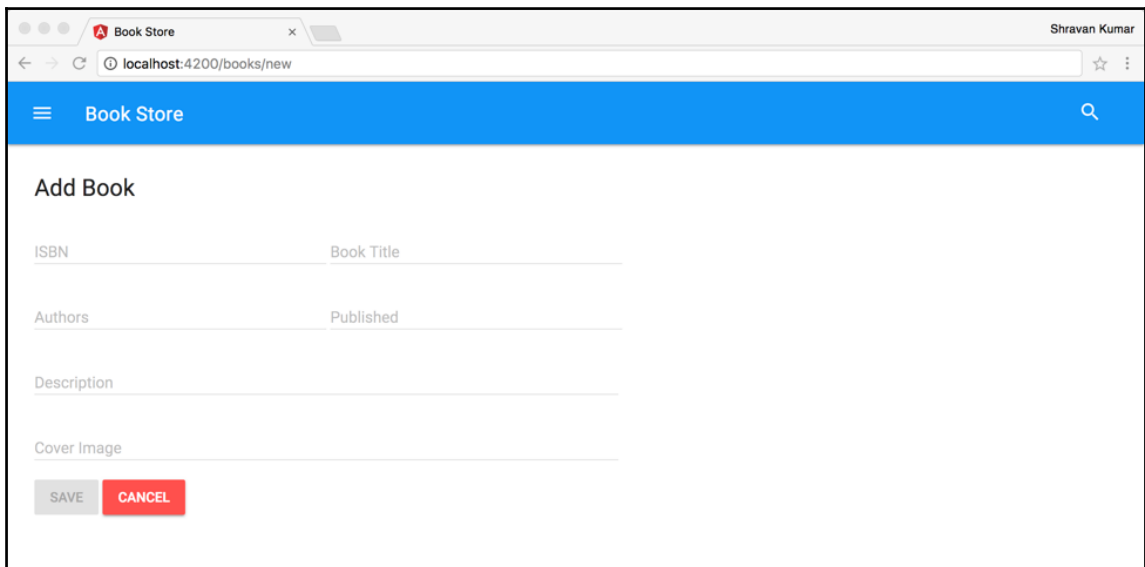
Animating routed components

Motion adds more life to the user interface when it is carefully implemented. Animations let us add different kinds of motions to the applications to make UI more appealing.

Angular implemented an animation system on top, **Web Animations API**, and it lets us build animations that run at native performance, such as pure CSS animations. The browsers that do not support the Web Animations API yet need the `web-animations.min.js` polyfill.

For more information on Web Animations API, visit `https://w3c.github.io/web-animations`. The `web-animations.min.js` polyfill file can be downloaded at `https://github.com/web-animations/web-animations-js`.

In this section, we are going to learn how to animate while navigating between the components.

First, let us see how to add the `AnimationsModule` to `AppModule`. The code for `src/app/app.module.ts` is as follows:

```
import { BrowserAnimationsModule } from
              '@angular/platform-browser/animations';

...

@NgModule({
  ...
  imports: [
    ...
    BrowserAnimationsModule,
    RouterModule.forRoot(routes)
  ]
  ...
})
export class AppModule {
}
```

Now we will define animations. The code for `src/app/animations.ts` is as follows:

```
import { animate, state, style, transition, trigger,

AnimationTriggerMetadata } from '@angular/animations';

export const slideInOutAnimation: AnimationTriggerMetadata =
  trigger('routeAnimation', [
    state('*',
```

```
    style({
      opacity: 1,
      transform: 'translateX(0)'
    })
  ),
  transition(':enter', [
    style({
      opacity: 0,
      transform: 'translateX(-100%)'
    }),
    animate('0.2s ease-in')
  ]),
  transition(':leave', [
    animate('0.4s ease-out', style({
      opacity: 0,
      transform: 'translateX(100%)'
    }))
  ])
]);
```

We use the following methods to define animations:

- `trigger()`: This creates an animation trigger with a list of states and transition
- `state()`: This declares an animation state within the given trigger; we are using the `*` in our code, and it matches any animation state
- `style()`: This takes in a key/value pair of CSS property/value pairs
- `transition()`: This declares animation steps

We are creating animations for the component while entering and leaving the route state. While entering, our component animates from left to right, while leaving, it animates from right to left.

After defining animations, add animations to the component:

The code for `src/app/books/books-list.component.ts` is as follows:

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

import { slideInOutAnimation } from '../../animations';

@Component({
  ...
  animations: [slideInOutAnimation]
})
export class BookDetailsComponent implements OnInit {
  ...
```

```
    @HostBinding('@routeAnimation') routeAnimation = true;
    @HostBinding('style.display')   display = 'block';
    @HostBinding('style.position')  position = 'absolute';
    ...
}
```

We imported the animation defined in the previous step and added it to the `animations` array in the `@Component()` decorator, and we access the animation trigger and styles using the `@HostBinding()` decorator.

We can follow the earlier steps to add the animation to any component in our example.

Feature modules using @NgModule()

As the number of components increases in the application, it becomes complex, and we should segregate our components into different modules based on their functionality to manage the complexity. Let us understand how to use `@NgModule()` to structure our application components into feature modules.

We have only one module in our application; let us refactor it to create one more module. We have a lot of functionality-related books in our application, so let's create a separate module for this.

In the modules, we need to create a separate module for routes, and it keeps our feature module class clean. Here is the books routing module, which includes all the book-related routes.

The code for `src/app/books/books-routing.module.ts` is as follows:

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

import { BooksListComponent } from
'./books-list/books-list.component';
import { BookDetailsComponent } from
'./book-details/book-details.component';
import { NewBookComponent } from
'./new-book/new-book.component';

const routes: Routes = [
  {path: 'books', component: BooksListComponent},
  {path: 'books/new', component: NewBookComponent},
  {path: 'books/:id', component: BookDetailsComponent}
];
```

```
@NgModule({
  imports: [
    RouterModule.forChild(routes)
  ],
  exports: [
    RouterModule
  ]
})
export class BooksRoutingModule {
}
```

In the books routing module, while adding the routes to the imports array, we use `forChild()` because this is going to be a child of the main application module.

Let us create the books feature module and add all the related components, services, and routes.

The code for `src/app/books/books.module.ts` is as follows:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { BooksListComponent } from
'./books-list/books-list.component';
import { BookDetailsComponent } from
'./book-details/book-details.component';
import { NewBookComponent } from
'./new-book/new-book.component';

import { BookStoreService } from './book-store.service';
import { BooksRoutingModule } from './books-routing.module';

@NgModule({
  declarations: [
    BooksListComponent,
    BookDetailsComponent,
    NewBookComponent
  ],
  imports: [
    CommonModule,
    ReactiveFormsModule,
    HttpModule,
    BooksRoutingModule
  ],
  providers: [BookStoreService]
})
```

```
export class BooksModule {
}
```

By now, we have a separate feature module for the books, and we need to add the main `AppModule` before that lets us define a separate route module for `AppModule`.

The code for `src/app/app-routing.module.ts` is as follows:

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

import { DashboardComponent } from './dashboard.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  {path: '', redirectTo: 'dashboard', pathMatch: 'full'},
  {path: 'dashboard', component: DashboardComponent},
  {path: 'about', component: AboutComponent}
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {
}
```

Now we need to include the `AppRoutingModule` and books feature module to the `AppModule`:

The code for `src/app/app.module.ts` is as follows:

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

import { AppComponent } from './app.component';
import { DashboardComponent } from './dashboard.component';
import { AboutComponent } from './about.component';
import { Safe } from './safe';

import { AppRoutingModule } from './app-routing.module';
import { BooksModule } from './books/books.module';

@NgModule({
```

```
    declarations: [
      AppComponent,
      DashboardComponent,
      AboutComponent,
      Safe
    ],
    imports: [
      BrowserModule,
      BooksModule,
      AppRoutingModule
    ],
    bootstrap: [AppComponent]
})
export class AppModule {
}
```

If we look at the `AppModule`, it looks tiny and clean now. Depending on the application size, we create as many modules as we need. With the features, we can use the features like lazy loading and preloading to improve the performance of the application also.

The source code for the refactored example is available under the `Chapter6/book-store-extended` folder.

Summary

We started this chapter by discussing how to communicate with REST services using an HTTP client, and we developed a basic example. Then, we refactored all the HTTP client-related code to a service. Then, you learned about Angular routing basics and then we implemented all the features in a Book Store application. We looked how to add animation to routed components; finally, you learned how to refactor our application into features modules.

By the end of this chapter, the reader should have a good understanding of how to build any UI application with various Angular features, such as components, forms, HTTP, and routing. In the next chapter, we will discuss how to test the Book Store application we created in this chapter.

7
Testing

In this chapter, you will learn how to test Angular applications using different kinds of testing techniques and tools. We look at some basic examples and real-world examples. After going through this chapter, the reader will understand the following concepts:

- Unit testing and end-to-end testing
- How to write isolated and integrated unit tests
- How to unit test components and services

Testing

Testing is one of the important aspects of application development, which ensures that the application is working fine before we deploy to production for end user usage; it helps find the bugs early and also ensures that we won't break the existing functionality as we add new features to the application. It is important to make it a part of the development process itself.

There are different kinds of software development processes that focus on testing. The **test-driven development** (**TDD**) is one kind of technique that emphasizes on writing the tests first then the actual functionality; we won't be diving more into TDD, which is beyond the scope of this book. This chapter focuses on the following two primary tests methodologies used by developers during the development:

- Unit testing
- End-to-end testing

Unit testing

Unit testing focuses on testing the individual parts of the applications; for example, in the Angular application, we unit test functionality inside the components, services, directives, and pipes.

End-to-end testing

End-to-end testing focuses on testing the entire application, and these tests run against the application running in a real browser, interacting with it as a user would in the real world. In this chapter, we are covering unit testing, end-to-end testing is beyond the scope of this book.

Before we start writing the test, let's look at the tools required for unit testing.

Tooling

The following are the tools required for testing:

- **Jasmine**: Jasmine is a behavior-driven development framework in order to test JavaScript code, you can find more information on Jasmine at: `https://jasmine.github.io`
- **Karma**: Karma is a test runner we use for running our unit test while developing; you can find more information on Karma at: `http://karma-runner.github.io`
- **Protractor**: Protractor is an end-to-end testing framework for Angular applications. You can find more information on Protractor at: `http://protractortest.org`

Configuration files

The following are the Karma configuration files:

- **karma.conf.js**: This is the Karma configuration file that specifies which plugins to use, which application and test files to load, which browser(s) to use, and how to report test results.
- **karma-test-shim.js**: This is the shim that makes Karma work with the Angular test environment and launches Karma itself; it includes some of the SystemJS configuration in order to load Angular test utilities.

Jasmine basics

Before we start writing the unit tests, let's look at some Jasmine functions that we use for writing every unit test.

- `describe()`: The describe function is a global Jasmine function. It is used for grouping similar kinds of tests/specs together in a suite. The describe functions can be nested. The syntax is as follows:

```
describe('suite name', () => {

//unit tests – it functions...
});
```

- `it()`: It is a Jasmine function used for writing the actual unit tests. The syntax is as follows:

```
it('test name', () => {
//unit test code
});
```

- **Matchers**: Matchers are the built-in Jasmine functions used along with the **expect()** function to compare the actual value with the expected value. Here are the matcher functions provided by Jasmine:
    - `toBe()`
    - `toEqual()`
    - `toMatch()`
    - `toBeDefined()`
    - `toBeUndefined()`
    - `toBeNull()`
    - `toBeNaN()`
    - `toBeTruthy()`
    - `toBeFalsy()`
    - `toHaveBeenCalled()`
    - `toHaveBeenCalledWith()`
    - `toHaveBeenCalledTimes()`
    - `toContain()`
    - `toBeLessThan()`
    - `toBeLessThanOrEqual()`

[156]

- `toBeGreaterThan()`
- `toBeGreaterThanOrEqual()`
- `toBeCloseTo()`
- `toThrow()`
- `toThrowError()`

- `expect()`: It is an another Jasmine function that takes a value named actual value, and it is used along with matcher functions to assert the expected value.
- `beforeEach()`: `beforeEach()` is a Jasmine built-in function that runs the code inside it before every test in the `describe()` function.
- `afterEach()`: `afterEach()` is a Jasmine built-in function that runs the code inside it after every test in the `describe()` function.
- `beforeAll()`: `beforeAll()` is a Jasmine built-in function that runs the code inside it only once before all the tests in the `describe()` function.
- `afterAll()`: `afterAll()` is a Jasmine built-in function that runs the code inside it only once after all tests completes the execution in the `describe()` function.

Unit testing

We can write two kinds of unit tests for Angular applications:

- Isolated unit tests
- Integrated unit tests

Isolated unit tests instantiate the class directly inside tests without any dependency on Angular. They are used for testing only the component's logic (not the template), and they are suitable for testing services, pipes, and directives.

Integrated unit tests are written using Angular test utility classes; they are used for testing more complex scenarios dependent on Angular features, such as modules and templates.

Isolated unit tests

In this section, you are going to learn how to use some basics isolated unit tests. We are going to use the `unit-testing-setup` project in the provided source code; this is just a *hello world* Angular application. We are going to install Karma, Jasmine npm packages, and set up the Karma to run our test using the Jasmine framework.

Let's create a project named `01-isolated-unit-tests` from the `unit-testing-setup` project. First, we need to install the npm packages and run the following commands in the project:

```
npm install jasmine-core jasmine --save-dev
npm install karma karma-cli --save-dev
npm install karma-jasmine karma-chrome-launcher
```

Now add the following code to the scripts section in the `package.json` file to run Karma directly using the `npm run` command:

```
"karma": "karma start karma.conf.js",
"pretest:once": "npm run build",
"pretest": "npm run build",
"test:once": "npm run karma -- --single-run",
"test": "concurrently \"npm run build:watch\" \"npm run karma\""
```

We need to include the `karma.conf.js` and `karma-test-shim.js` files to the root of our project. The following is an example Karma configuration file, which provides the instructions to the Karma test runner for which testing framework we want to use, required plugins to run the tests, what files to include in the test, and what to exclude:

```
module.exports = function (config) {
  var appSrcBase = 'src/';
  var appAssets = '/base/app/';
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher')
    ],
    client: {
      builtPaths: [appSrcBase]
    },
    files: [],
    proxies: {},
    exclude: [],
    preprocessors: {},
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    concurrency: Infinity
```

```
        });

    }
```

Our original `karma.conf.js` and `karma-test-shim.js` files in the project are very lengthy; you can find them in the provided source code.

Writing basic isolated unit tests

Before we start writing unit tests, let's verify our test setup. Add the following unit test to the project (`src/app/app.component.spec.ts`):

```
describe('my first unit test', () => {
  it('true is true', () => expect(true).toBe(true));
});
```

Now go to the command line, run the following command:

```
npm run test:once
```

The preceding command will run the test that we added in the previous step. If our setup is ok at the end, we will get `Executed 1 of 1 SUCCESS` message and Karma will terminate the execution.

```
02 04 2017 23:58:03.824:INFO [karma]: Karma v1.5.0 server started at http://0.0.0.0:9876/
02 04 2017 23:58:03.825:INFO [launcher]: Launching browser Chrome with unlimited concurrency
02 04 2017 23:58:03.836:INFO [launcher]: Starting browser Chrome
02 04 2017 23:58:04.582:INFO [Chrome 57.0.2987 (Mac OS X 10.12.4)]: Connected on socket kyD5x8aIlwE9XW82AAAA with id 96036093
02 04 2017 23:58:04.797:WARN [web-server]: 404: /base/src/systemjs.config.extras.js
Chrome 57.0.2987 (Mac OS X 10.12.4): Executed 1 of 1 SUCCESS (0.005 secs / 0.002 secs)
```

Before we proceed, I want to discuss a bit about the test file name. It is using the same name as the component name with the `.spec` suffix. In the Jasmine framework, tests are named specs; it is a general convention to suffix all the test files with `.spec` and use the same filename (components, services, directives, pipes, and routes), which we are testing.

Once again, go to the command line, run the following command:

```
npm run test
```

The preceding command will run Karma in watch mode. Every time we make a change in the source code or test code, Karma will automatically run all the unit tests again.

Let's write some unit tests for our `AppComponent`:

The code for `src/app/app.component.spec.ts` is as follows:

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

describe('AppComponent', () => {

  it('name is initialized with Angular', () => {
    let component = new AppComponent();
    expect(component.name).toBe('Angular');
  });

  it('name to be Angular UI', () => {
    let component = new AppComponent();
    expect(component.name).toBe('Angular');

    component.name = 'Angular UI';
    expect(component.name).toBe('Angular UI');
  });

});
```

We have two unit tests, one is checking the initial value of the `name` property in the `AppComponent` class, and another one is verifying the changes to the `name` property. In both the tests, we are instantiating the `AppComponent` class, which is not necessary, that we can use the `beforeEach()` method in Jasmine framework to run the same code before every test:

The code for `src/app/app.component.spec.ts` is as follows:

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

describe('AppComponent', () => {
  let component: AppComponent;

  beforeEach(() => {
    component = new AppComponent();
  });

  it('name is initialized with Angular', () => {
    expect(component.name).toBe('Angular');
  });

  it('name to be Angular UI', () => {
    expect(component.name).toBe('Angular');
```

```
        component.name = 'Angular UI';
        expect(component.name).toBe('Angular UI');
    });

  });
```

We understood how to write basic unit tests, but our `AppComponent` class does not have any real functionality that we can test. Let's use the bookstore example that we developed in the previous chapter so that we can understand how to write some useful unit tests.

You can use the `book-store-start` application in the `Chapter7\book-store-start` source code to get started. This application is created using the Angular CLI, so it has all necessary configuration in it already. Let's create an `book-store` application from `book-store-start` application.

Testing services

In our Book Store application, we have `BookStoreService`, which communicates with the external REST service using the Angular HTTP service to perform different operations on the books list:

The code for `src/app/books/book-store.service.spec.ts` is as follows:

```
import { BookStoreService } from './book-store.service';

describe('BookStoreService', () => {
  let bookStoreService: BookStoreService;

  beforeEach(() => {
    bookStoreService = new BookStoreService();
  });
});
```

The preceding code snippet is incomplete. The `BookStoreService` constructor is expecting an Angular HTTP service object as a parameter, and this is required because of the methods in our service using HTTP methods, such as `get()`, `post()`, and `delete()` for different operations. However, we should not call the real REST service using HTTP because we want to test our service behavior, not the external REST service.

In these scenarios, we should mock the required object, and this can be simply done using the `jasmine.createSpyObj()` method.

Mocking dependencies

Let's mock the Angular HTTP service using the `jasmine.createSpyObj()` method:

The code for `src/app/books/book-store.service.spec.ts` is as follows:

```
import { BookStoreService } from './book-store.service';

describe('BookStoreService', () => {
  let bookStoreService: BookStoreService,
    mockHttp;

  beforeEach(() => {
    mockHttp = jasmine.createSpyObj('mockHttp',
                                    ['get', 'post', 'delete']);

    bookStoreService = new BookStoreService(mockHttp);
  });
});
```

The `jasmine.createSpyObj()` method takes the mock object name as the first parameter and methods for the mock object in the second parameter as an array.

Here is the test for the `deleteBook()` method in `BookStoreService`:

```
it('deleteBook should remove the book', () => {
  const book: Book = {
    id: 12,
    isbn: 9781849692380,
    title: 'test title',
    authors: 'test author',
    published: 'test date',
    description: 'test description',
    coverImage: 'test image'
  };

  mockHttp.delete.and.returnValue(Observable.of(book));
  const response = bookStoreService.deleteBook(12);
  response.subscribe(value => {
    expect(value).toBe(book);
});
```

The `deleteBook()` method returns the book we delete as an Observable and we are mocking that return value using the Jasmine `returnValue()` method. We are using the `subscribe()` method to receive the values and comparing the response value with book.

---

[162]

Let's write some more unit tests and verify the parameters passed to the HTTP `delete` method. The code for `src/app/books/book-store.service.spec.ts` is as follows:

```
import { BookStoreService } from './book-store.service';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import { Book } from './book';

describe('BookStoreService', () => {
  let bookStoreService: BookStoreService,
    mockHttp;

  beforeEach(() => {
    mockHttp = jasmine.createSpyObj('mockHttp', ['get', 'post', 'delete']);
    bookStoreService = new BookStoreService(mockHttp);
  });

  describe('deleteBook', () => {

    it('should remove the book', () => {
      const book: Book = {
        id: 12,
        isbn: 9781849692380,
        title: 'test title',
        authors: 'test author',
        published: 'test date',
        description: 'test description',
        coverImage: 'test image'
      };

      mockHttp.delete.and.returnValue(Observable.of(book));
      const response = bookStoreService.deleteBook(12);
      response.subscribe(value => {
        expect(value).toBe(book);
    });

    it('should call http delete method with right url', () => {
      const id = 12;
      const url = `http://58e15045f7d7f41200261f77.mockapi.io/
api/v1/books/${id}`;
      mockHttp.delete.and.returnValue(Observable.of(true));
      const response = bookStoreService.deleteBook(id);
      expect(mockHttp.delete).toHaveBeenCalledWith(url,
                                         jasmine.any(Object));
    });

  });
});
```

We can test the remaining methods similarly. Let's look at how to test components using isolated unit tests.

Testing components

In our Book Store application, we have multiple components. Let's look at how to test `BooksListComponent`. Testing the component is very similar to the way we tested service.

The code for `src/app/books/books-list/books-list.component.spec.ts` is as follows:

```
import { BooksListComponent } from './books-list.component';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';

describe('BooksListComponent', () => {
  let booksListComponent: BooksListComponent,
    mockBookStoreService;

  beforeEach(() => {
    mockBookStoreService = jasmine.createSpyObj(
                         'mockBookStoreService', ['getBooks']);
    booksListComponent =
               new BooksListComponent(mockBookStoreService);
  });

  it('initial books list should be empty', () => {
    expect(booksListComponent.booksList.length).toBe(0);
  });

  describe('ngOnInit', () => {

    it('should fetch books list', () => {
      const books = [{}, {}];
      expect(booksListComponent.booksList.length).toBe(0);
      mockBookStoreService.getBooks
                       .and.returnValue(Observable.of(books));
      booksListComponent.ngOnInit();
      expect(booksListComponent.booksList.length).toBe(2);
    });

  });
});
```

Our `BooksListComponent` is dependent on `BookStoreService`, so we need to mock this. We have a `booksList` property, which is initially empty after we invoke the `ngOnInit()` method. The `booksList` property might change; we need to test this behavior. In the application, `ngOnInit()` is invoked as part of the component life cycle; here, we need to invoke it explicitly.

Integrated unit tests

Testing simple bindings and method logic is enough most of the times, but we also want to understand how our component logic is working along with the templates, child components, and routes. To do this, isolated unit tests are enough.

Testing a component with a simple template also could be complex. For this, Angular provides testing utilities in the `@angular/core/testing` module. These testing utility classes help us to test our application close to the Angular runtime environment.

Testing components

Here is our `AboutComponent`, which just displays the property values on the template; this is the right place to start writing some integrated unit tests:

The code for `src/app/about.component.ts` is as follows:

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

@Component({
  selector: 'about-page',
  template: `
    <div>
      <h4>{{heading}}</h4>
      <p class="message">{{content}}</p>
    </div>
  `,
})
export class AboutComponent {
  heading = 'This is About Page';
  content = '';
}
```

The code for `src/app/about.component.integrated.spec.ts` is as follows:

```
import { ComponentFixture, TestBed }
                             from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { AboutComponent } from './about.component';

describe('AboutComponent', () => {
  let component: AboutComponent;
  let fixture: ComponentFixture<AboutComponent>;
  let debugElement: DebugElement;
  let element: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AboutComponent]
    });

    fixture = TestBed.createComponent(AboutComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement.query(By.css('h4'));
    element = debugElement.nativeElement;
  });

  it('should display "This is About Page"', () => {
    fixture.detectChanges();
    expect(element.textContent).toContain(component.heading);
  });
});
```

The preceding integrated unit test is just checking for the heading value in `AboutComponent`. It is lengthy compared to similarly isolated unit tests because we are testing it in the proximate Angular runtime environment.

Let us understand the test line by line. First, we imported all required Angular testing utility classes, then the component we need to unit test:

- `TestBed`: This creates the environment for testing Angular applications (creating Angular modules and components for testing).
- `ComponentFixture`: This is fixture for testing the components; this provides the properties and methods to access the component instance, DOM elements inside the component's template, and run the change detection manually.
- `DebugElement`: This provides access to the root element of the component.
- `HTMLElement`: This represents the native DOM HTML element.

Now we can write the integrated unit tests using the Jasmine methods `describe()`, `beforeEach()`, and `it()`.

First, we are configuring our test module using the `configureTestingModule()` method in the `TestBed` class, which is similar to `@NgModule()` and takes an object as a parameter with the following properties: `providers`, `declarations`, `import`, and `schemas`.

Then, we are creating the component that returns a fixture for accessing the component instance. Once we have an access to the component instance; we query using root element using the `DebugElement` class `query()` method. To the `query()` method, we need to pass a predicate; it is passed by a CSS selector using the `By.css()` method.

> The `By.css()` method matches the elements by the given CSS selector.

The `DebugElement` class `query()` method returns the first element matched by the selector, and we can get all elements using the `queryAll()` method.

The `By` class in the `@angular/platform-browser` module provides two more methods for accessing elements:

- `By.all()`: This matches all elements
- `By.directive()`: This matches elements that have the given directive present

The native DOM element is accessed using the `nativeElement` property of the `DebugElement`, using which we can access the test and child elements inside of it.

Finally, we are comparing the component heading value to the text on the template. However, there is one interesting thing in our test `detectChanges()` method; Angular won't run the change detection automatically in the test environment, we need to use the `detectChanges()` method every time we modify the data.

Let's add a couple of more tests for different scenarios.

The code for `src/app/about.component.integrated.spec.ts` is as follows:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [AboutComponent]
  });

  fixture = TestBed.createComponent(AboutComponent);
```

```
    component = fixture.componentInstance;
  });

  describe('heading', () => {
    beforeEach(() => {
      debugElement = fixture.debugElement.query(By.css('h4'));
      element = debugElement.nativeElement;
    });

    it('should display "This is About Page"', () => {
      fixture.detectChanges();
      expect(element.textContent).toContain(component.heading);
    });

    it('should display "new heading"', () => {
      fixture.detectChanges();

      const previousHeading = component.heading;
      component.heading = 'new heading';

      expect(element.textContent).toContain(previousHeading);
      expect(element.textContent)
                           .not.toContain(component.heading);

      fixture.detectChanges();
      expect(element.textContent).toContain(component.heading);
    });
  });

  describe('content', () => {
    beforeEach(() => {
      debugElement = fixture
                       .debugElement.query(By.css('.message'));
      element = debugElement.nativeElement;
    });

    it('should be empty', () => {
      fixture.detectChanges();
      expect(element.textContent).toBe(component.content);
    });

    it('should be "new message"', () => {
      component.content = 'new message';
      fixture.detectChanges();
      expect(element.textContent).toBe(component.content);
    });
  });
```

In the second test, we are checking for the modified heading value, and in the third and fourth tests, we are testing the content property values.

The `AboutComponent` has an inline template. If it has an external template or external style sheets, the previously-mentioned unit tests won't work. Angular downloads these files asynchronously, but our unit tests are running synchronously. We can use the `async()` method in the `@angular/core/testing` module to handle asynchronous operations in our tests; here is the example unit test of `AboutComponent` with an external template:

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [AboutComponent]
  });
}));

beforeEach(() => {
  fixture = TestBed.createComponent(AboutComponent);
  component = fixture.componentInstance;
  debugElement = fixture.debugElement.query(By.css('h4'));
  element = debugElement.nativeElement;
});
```

We just need to wrap the test module creation in the `async()` method in a separate`beforeEach()` and the rest of the code in different blocks depending on the unit tests.

We need to chain the `configureTestingModule().compileComponents()` method to compile the templates and CSS files if we are using SystemJS. In our application, we are using the Angular CLI which internally uses webpack does this for us.

Testing components with dependencies

Up till now, we have tested a simple component with two properties, but components become complex with dependencies such as services, other components, child components, routes, and forms.

Let's take a look at `BooksListComponent`, which is dependent on the `BookStoreService`, and the `BookStoreService` method returns the Observables as results, and our template uses router directives.

We already know how to deal with dependent services using Jasmine's `spy()` methods. In the earlier section, we looked at how to write integrated unit test. Let's combine these two concepts to test the `BooksListComponent`:

The code for `src/app/books/books-list/books-list.component.integrated.spec.ts` is as follows:

```
import { ComponentFixture, TestBed } from
                                '@angular/core/testing';
import { async, ComponentFixtureAutoDetect } from
                                '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';
import { Observable } from 'rxjs/Observable';

import { BooksListComponent } from './books-list.component';
import { BookStoreService } from '../book-store.service';
import { Book } from '../book';

describe('BooksListComponent', () => {
  let fixture: ComponentFixture<BooksListComponent>,
    component: BooksListComponent,
    debugElement: DebugElement,
    element: HTMLElement,
    mockBookStoreService;

  const booksList: Book[] = [{
    id: 1,
    isbn: 9781783980628,
    title: 'Getting Started with Grunt',
    authors: 'Jaime Pillora',
    published: 'February 2014',
    description: 'JavaScript and Grunt.',
    coverImage: 'https://test.com/img1.png'
  }];

  beforeEach(async(() => {

    mockBookStoreService = jasmine
        .createSpyObj('mockBookStoreService', ['getBooks']);
    mockBookStoreService.getBooks
                .and.returnValue(Observable.of(booksList));

    TestBed.configureTestingModule({
      declarations: [
        BooksListComponent
      ],
      providers: [
        { provide: ComponentFixtureAutoDetect,
                                        useValue: true },
        { provide: BookStoreService,
                        useValue: mockBookStoreService }
```

[170]

```
        ]
    });
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BooksListComponent);
    component = fixture.componentInstance;
  });
});
```

We have the initial setup for `BooksListComponent`. We mocked the `BookStoreService` method's `getBooks()` method to return a dummy `Observable` of books. We also set `ComponentFixtureAutoDetect` to `true`, and this will run the initial change detection automatically in every test.

Here is our first test checking value returned from the `mockBookStoreService`:

```
it('should display books list', () => {
  debugElement = fixture.debugElement
                            .query(By.css('.book-card'));
  element = debugElement.nativeElement.firstElementChild;
  expect(element.style.backgroundImage)
                    .toContain(booksList[0].coverImage);
});
```

Here is our second test testing the component values without change detection:

```
it('should not display updated books list', () => {

  component.booksList = [{
    id: 2,
    isbn: 9781786462084,
    title: 'Laravel 5.x Cookbook',
    authors: 'Alfred Nutile',
    published: 'September 2016',
    description: 'Laravel 5.x',
    coverImage: 'https://test.com/img2.png'
  }];

  debugElement = fixture.debugElement
                            .query(By.css('.book-card'));
  element = debugElement.nativeElement.firstElementChild;

  expect(element.style.backgroundImage)
                        .toContain(booksList[0].coverImage);
  expect(element.style.backgroundImage)
            .not.toContain(component.booksList[0].coverImage);
});
```

In the earlier-mentioned test without calling the `detectChanges()` method, the component still uses the old values after assigning the `booksList` with the new set of books.

Here is our third test to check the updated values with change detection:

```
it('should display updated books list', () => {
  component.booksList = [{
    id: 2,
    isbn: 9781786462084,
    title: 'Laravel 5.x Cookbook',
    authors: 'Alfred Nutile',
    published: 'September 2016',
    description: 'Laravel 5.x',
    coverImage: 'https://test.com/img2.png'
  }];

  fixture.detectChanges();

  debugElement = fixture.debugElement
                              .query(By.css('.book-card'));
  element = debugElement.nativeElement.firstElementChild;

  expect(element.style.backgroundImage)
              .toContain(component.booksList[0].coverImage);
});
```

Summary

We started this chapter by discussing different testing mechanisms, and you learned why testing is necessary. We looked at the different unit testing strategies to test Angular code.

You learned how to write isolated unit tests and integrated unit tests for various parts (services, components, and so on) of the Angular application. By the end of this chapter, a user should have a good understanding of how to write the unit tests for Angular applications.

8

Angular Material

In this chapter, we will learn how to develop the visually compelling application using Angular Material components. We look at different UI controls provided by Angular Material and how to use them in various scenarios. After going through this chapter, the reader understands the following concepts:

- Material design
- How to use Material design components

Introduction

Angular Material is a set of high-quality UI components developed by the Angular team-based Google Material design specification. These UI components help us to build a single, good-looking UI that spans across multiple devices.

Getting started

In this chapter, you are going to learn how to use UI components provided by Angular Material to build the applications. Instead of looking at individual controls, we are going to develop a complete application using these components.

In `Chapter 6`, *Building a Book Store Application*, we built a Book Store application using **Material Design Lite**, where we wrote a lot of boilerplate code to make our application look good. We are going to develop the same application using Angular Material; learn how it helps to achieve similar functionality with less code to make a better-looking application.

> Material Design Lite is also based on the Google Material design specification only; it does not rely on any JavaScript frameworks.

Project setup

The following are the steps to include Angular Material into our Book Store application. We can use the `book-store-start` application under `Chapter8` source code to get started with the setup.

Let us first install Angular Material. The following command will install Angular Material:

```
npm install @angular/material --save
```

Now we will include Angular animations into `AppModule`. Some of the Angular Material components depends on the Angular animations module for advanced transitions. Let's install and include it in our project.

```
npm install @angular/animations --save
```

Import it in our `AppModule` and add it to `@NgModule()` imports array:

```
import { BrowserAnimationsModule } from
              '@angular/platform-browser/animations';

@NgModule({
  ...
imports: [
    BrowserModule,
    HttpModule,
    BrowserAnimationsModule
  ],
  ...
})
export class AppModule { }
```

Now we will include a theme to `index.html` file.

We should include a theme for all Material component styles, under the `node_modules/@angular/material/prebuilt-themes` folder, we have the following four out-of-the-box themes available:

- `deeppurple-amber`
- `indigo-pink`
- `pink-bluegrey`
- `purple-green`

We can include any one of the preceding themes, or we can include our custom theme also. For our application, we are going to use the `indigo-pink` theme, so let's add it to our `index.html` file:

```
<link href="../node_modules/@angular/material/
            prebuilt-themes/indigo-pink.css" rel="stylesheet">
```

Let's add **HammerJS** for gesture support.

Some of the Angular Material components such as `MdTooltip` and `MdSlider` depend on HammerJS for gestures. We need to install and include it to our `AppModule`:

```
npm install hammerjs --save
```

Import it in our `AppModule`:

```
import 'hammerjs';
```

Angular Material setup for our application is done. Let us also include the *Roboto* font and *Material design icons* to our `index.html`. These are optional, and we can use any font or a different set of icons:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?
                            family=Roboto:300,400,500">

<link rel="stylesheet" href="https://fonts.googleapis.com/css?
                            family=Material+Icons">
```

Using Angular Material components

To begin with Angular Material first, we are going to develop a `master-detail` page. To host this page and other pages, we need a layout in our application.

We are going to use CSS flexbox for designing our layouts. Instead of writing a lot of CSS by hand, the Angular team developed a module named `@angular/flex-layout`, the flex layout module provided directives for using flexbox declaratively in Angular templates. We need to install and include it to our `AppModule`:

```
npm install @angular/flex-layout --save
```

Add it to `AppModule`:

```
import { FlexLayoutModule } from '@angular/flex-layout';

@NgModule({
  ...
  imports: [
    BrowserModule,
    HttpModule,
    BrowserAnimationsModule,
    FlexLayoutModule
  ],
  ...
})
```

> The Angular `FlexLayoutModule` can be used independently of Angular Material.

To learn about the flexbox layout, visit the following links:

- `https://css-tricks.com/snippets/css/a-guide-to-flexbox/`
- `https://github.com/angular/flex-layout/`

> At the time of writing this chapter, Angular Material is still in beta 3, and the APIs might change in future. Source code provided with the book will be updated to accommodate the latest changes in the framework.

Master-detail page

Let us create the `master-detail` page to show the books list and the selected book information from the books list.

The code for `src/app/books/master-detail/master-detail.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../book';
import { BookStoreService } from '../book-store.service';

@Component({
  selector: 'bl-master-detail',
  styleUrls: ['./master-detail.component.scss'],
  templateUrl: './master-detail.component.html'
})
export class MasterDetailComponent implements OnInit {
  booksList: Book[] = [];
  selectedBook: Book;

  constructor(private bookStoreService: BookStoreService) {
  }

  ngOnInit() {
    this.bookStoreService
      .getBooks()
      .subscribe(response => this.booksList = response);
  }
}
```

The component is exactly same as the previous chapter. All the material-related code is in the template. As we are going to build the `master-detail` page, we need the left container to display the list of books, right-side container to display the selected book information:

- We can use the `<md-sidenav>` to for the left-side container
- The `<md-sidenav>` and associated content live inside of an `<md-sidenav-container>`
- We can use a `div` for associated content inside `<md-sidenav-container>` to display right-side container
- The `<md-sidenav mode="side">` display the `sidenav` side-by-side with the right-side container.
- In the `<md-sidenav>`, we need to display the list of books, we can use `<md-list>` or `<md-nav-list>`:

The code for `src/app/books/master-detail/master-detail.component.html` is as follows:

```html
<div fxLayout="column" fxFlex>
  <h2 class="page-title">Books List Master Detail Page</h2>
  <md-sidenav-container flexLayout-="row" fxFlex
                                      class="books-list">
    <md-sidenav mode="side" opened>
      <md-nav-list>
        <md-list-item *ngFor="let book of booksList"
                               (click)="selectedBook = book">
          <img md-list-avatar [src]="book.coverImage" />
          <h2 md-line>{{book.title}}</h2>
          <p md-line> {{book.authors}} </p>
        </md-list-item>
      </md-nav-list>
    </md-sidenav>
    <div class="books-list-item" fxFlex>
      <div *ngIf="selectedBook" class="books-list-item--detail"
                          fxLayout="row" fxFlex.sm="column">
        <div class="books-list-item--coverimage">
          <img [src]="selectedBook.coverImage" />
        </div>
        <div class="books-list-item--content" fxFlex>
          <h3>{{selectedBook.title}}</h3>
          <p>{{selectedBook.authors}}</p>
          <p>{{selectedBook.published}}</p>
          <p>ISBN: {{selectedBook.isbn}}</p>
          <p>{{selectedBook.description}}</p>
        </div>
      </div>
    </div>
  </md-sidenav-container>
</div>
```

One last thing we need to do to make our component work, we are using Angular Material UI components such as `<md-sidenav>`, `<md-sidenav-container>`, and `<md-nav-list>`. Our application has no knowledge of these components, so we need to import and include their respective modules to the `AppModule`.

We are going to add material modules to separate module and include that module to AppModule, and this keeps our AppModule smaller and cleaner:

The code for src/app/app-material.module.ts is as follows:

```
import { NgModule } from '@angular/core';
import {
  MdSidenavModule,
  MdListModule
} from '@angular/material';

const MATERIAL_MODULES = [
  MdSidenavModule,
  MdListModule
];

@NgModule({
  imports: MATERIAL_MODULES,
  exports: MATERIAL_MODULES
})
export class AppMaterialModule { }
```

Any time we use a new Material component, its respective module should be added to AppMaterialModule:

The code for src/app/app.module.ts is as follows:

```
import { AppMaterialModule } from './app-material.module';

@NgModule({
  ...
  imports: [
    BrowserModule,
    HttpModule,
    BrowserAnimationsModule,
    FlexLayoutModule,
    AppMaterialModule
  ]
  ...
})
```

The following is the output of our `MasterDetailComponent`:
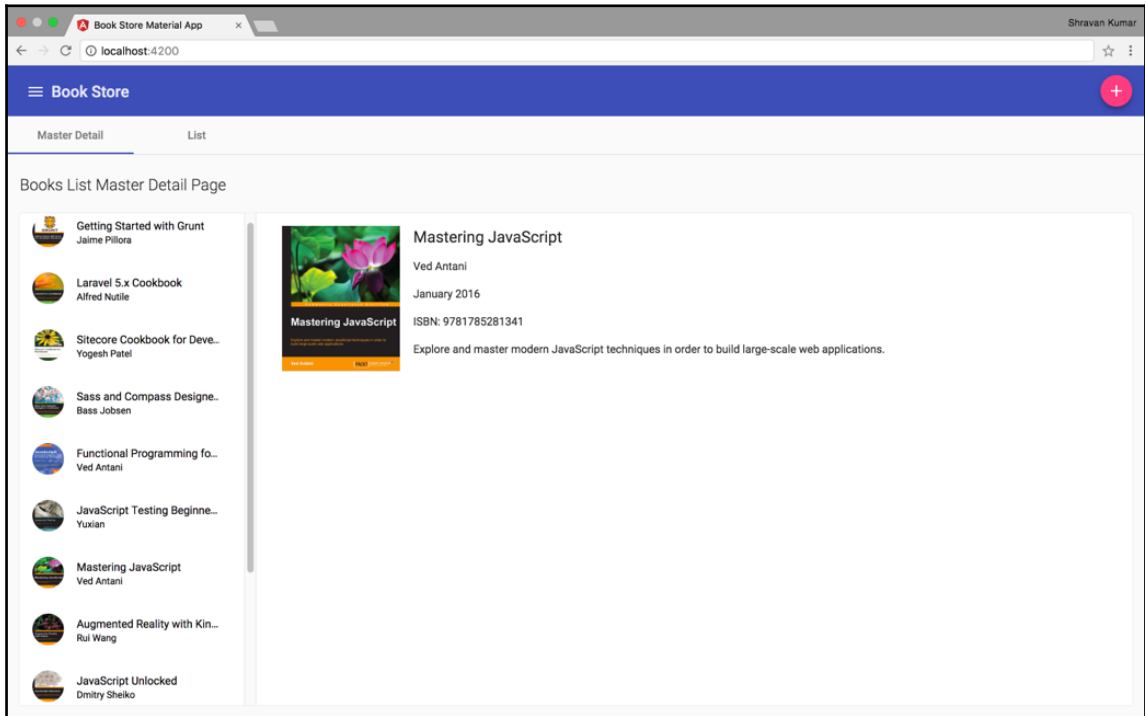
We cannot view the preceding output yet; for that, we need to use the component selector in our `AppComponent` template. We also need to add the `MasterDetailComponent` to our `AppModule` declarations array.

In `AppComponent`, we need a header to display the application title and other options; we also want to show the application navigation.

For the header, we can use `<md-toolbar>`; for navigation, once again use `<md-sidenav>`, `<md-nav-list>`. In the `AppComponent`, we are going to use `<md-tab-group>` to display multiple components based on the user selection of tabs:

The code for `src/app/app.component.ts` is as follows:

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

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

```
  })
  export class AppComponent {
    links = [{
      name: 'Books'
    }];
  }
```

The code for `src/app/app.component.html` is as follows:

```
<div fxLayout="column" fxFlex>
  <md-toolbar color="primary">
    <button md-icon-button (click)="sidenav.toggle()">
      <md-icon>menu</md-icon>
    </button>
    <span>Book Store</span>
  </md-toolbar>

  <md-sidenav-container fxFlex>
    <md-sidenav mode="over" #sidenav>
      <md-nav-list>
        <md-list-item *ngFor="let link of links">
          <p>{{link.name}}</p>
        </md-list-item>
      </md-nav-list>
    </md-sidenav>
    <div class="content" fxLayout="column" fxFlex>
      <md-tab-group>
        <md-tab label="Master Detail">
          <div fxFlex class="master-detail-container">
            <bl-master-detail fxFlex></bl-master-detail>
          </div>
        </md-tab>
        <md-tab label="List">
          LIST
        </md-tab>
      </md-tab-group>
    </div>
  </md-sidenav-container>
</div>
```

In the preceding template, the `sidenav` is hidden by default using `<md-sidenav mode="over">`. The over mode displays the `sidenav` on top all elements, and the rest of the screen is overlaid. We are displaying a menu icon in `<md-toolbar>` using `<md-icon>` inside a button that toggles the `sidenav` using the `toggle()` method of `<md-sidenav>`. We are invoking it using the `#sidenav` template reference variable.

We should also include `MdToolbarModule`, `MdButtonModule`, `MdIconModule`, and `MdTabsModule` in `AppMaterialModule`.

Start the application using the `npm start` command, and we can view the following output:
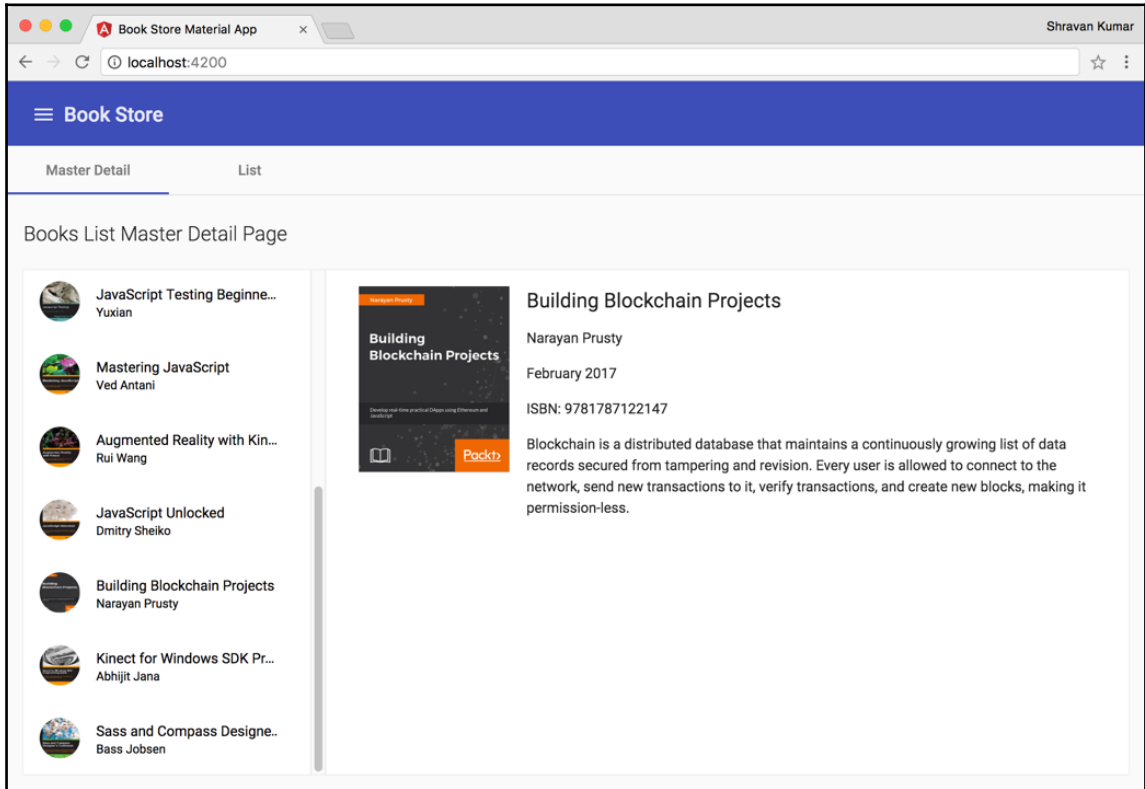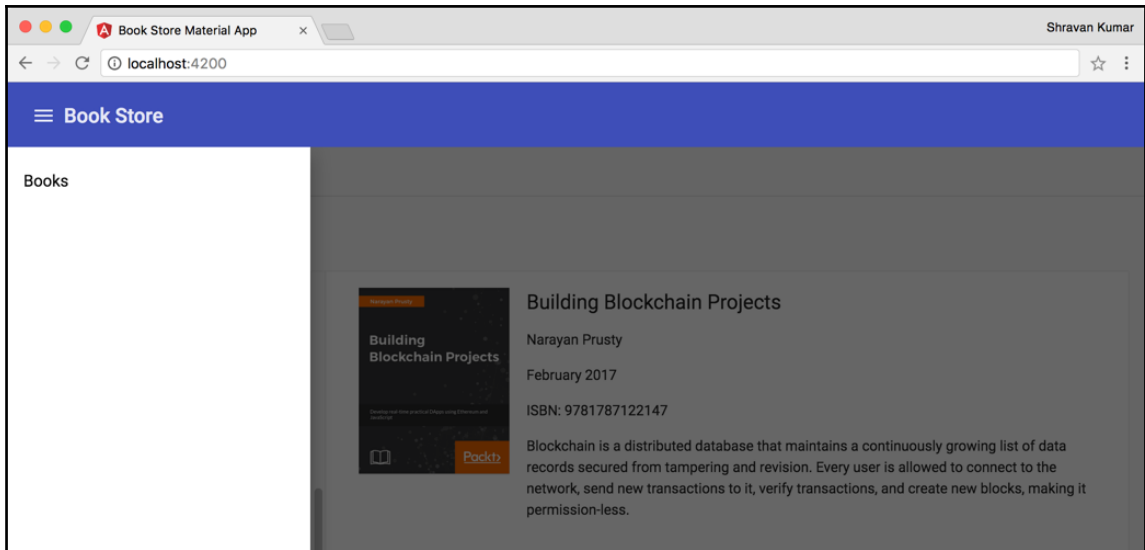
We successfully built our first two components using Angular Material. Let us create a different view to display the list of books using `<md-card>`.

Books list page

In this books list page, we are going to show books in the list view interface; for that, we are going to flexbox, and inside we will use `<md-card>` to display the books:

The code for `src/app/books/list/list.component.ts` is as follows:

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../book';
import { BookStoreService } from '../book-store.service';

@Component({
  selector: 'bl-list',
  styleUrls: ['./list.component.scss'],
  templateUrl: './list.component.html'
})
export class ListComponent implements OnInit {
  booksList: Book[] = [];

  constructor(private bookStoreService: BookStoreService) {
  }

  ngOnInit() {
```

```
    this.getBooks();
  }

  getBooks() {
    this.bookStoreService
      .getBooks()
      .subscribe(response => this.booksList = response);
  }

  deleteBook(id: number) {
    this.booksList = this.booksList
                      .filter(book => book.id !== id);
    this.bookStoreService.deleteBook(id)
      .subscribe(result => this.getBooks());
  }
}
```

The earlier-mentioned component has logic to get the books and delete the book using the service.

The code for `src/app/books/list/list.component.html` is as follows:

```html
<div fxLayout="column" fxFlex *ngIf="booksList.length > 0">
  <h2 class="page-title">Books List Page</h2>
  <section fxLayout="row" fxLayoutWrap fxLayout.sm="column"
                    fxLayoutGap="24px" fxLayoutAlign="center">

    <md-card class="book-card" *ngFor="let book of booksList">
      <img md-card-image [src]="book.coverImage">
      <md-card-actions>
        <button md-button>DETAIL</button>
        <button md-button  (click)="deleteBook(book.id)">
DELETE
        </button>
      </md-card-actions>
    </md-card>

  </section>
</div>
```

The `<md-card>` has other sections such as `<md-card-title>`, `<md-card-subtitle>`, and `<md-card-footer>` to display extra information. We should include `MdcardModule` in `AppMaterialModule` and `ListComponent` in the `AppModule` declarations array.

Now we should use the `<bl-list>` selector in the `AppComponent` template inside `<md-tab label="List"></md-tab>` to show the component:

```
<md-tab label="List">
  <div fxFlex class="books-list-container">
    <bl-list fxFlex></bl-list>
  </div>
</md-tab>
```

Here is the output we can view in the browser:
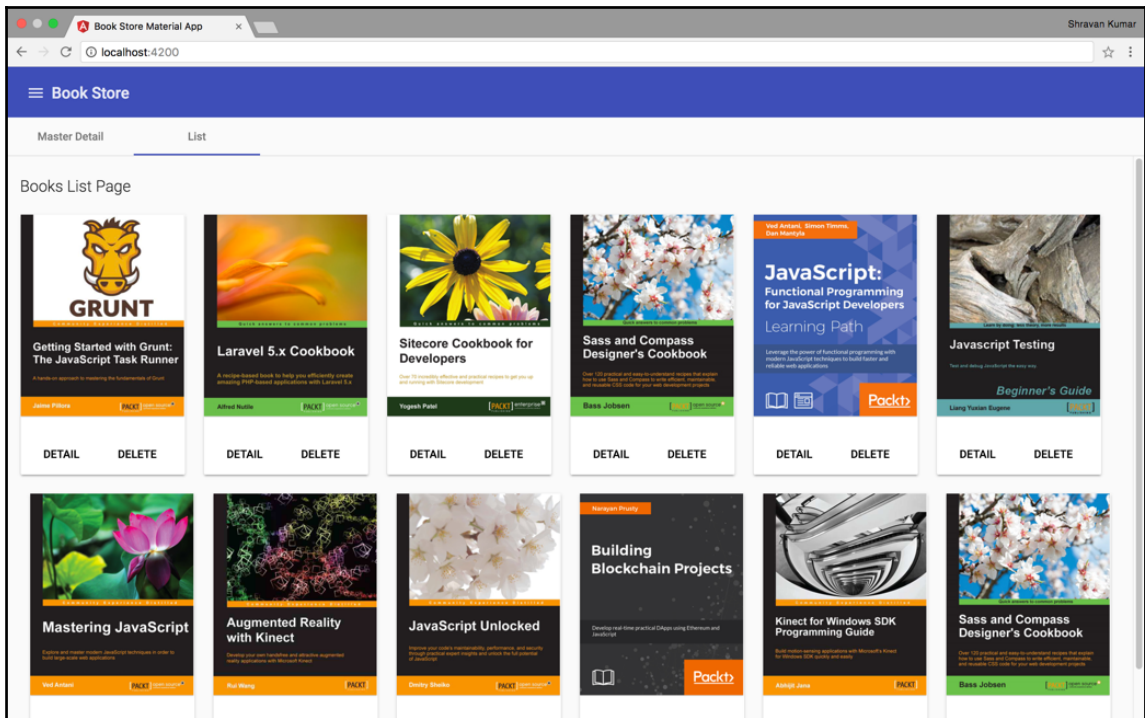
If we click on any of the **DELETE** buttons, it will remove the book and update the page, and it won't show any messages to the user. Let us use the `MdSnackBar` component to show the message when a book deletion is successful.

To use the `MdSnackBar` component, it needs to be injected into the constructor:

```
import { MdSnackBar } from '@angular/material';

constructor(
private bookStoreService: BookStoreService,
   private snackBar: MdSnackBar
) { }
```

We can use the `MdSnackBar` method's `openFromComponent()` method or the `open()` method to show it:

```
deleteBook(id: number) {
    this.booksList = this.booksList
                      .filter(book => book.id !== id);
    this.bookStoreService.deleteBook(id)
      .subscribe(result => {
        if (result.ok) {
          this.openSnackBar();
        }
        this.getBooks();
      });
  }

  openSnackBar() {
    this.snackBar.open('Book Deleted', 'CLOSE', {
      duration: 1000
    });
  }
```

Here is the output of using the `MdSnackBar` component:
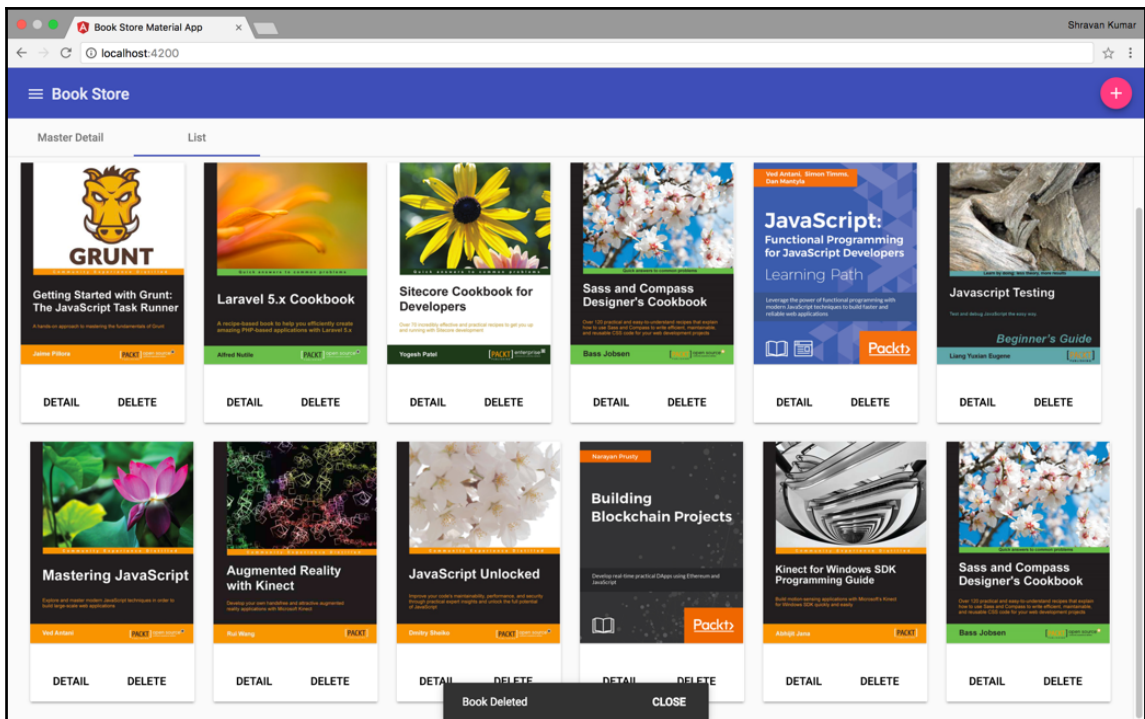
In our examples, we get the response quickly then we show the output immediately. However, in real-world scenarios, there will be a delay in getting the response, the screen will be blank, and the user does not understand what is happening. We can use `<md-progress-spinner>` to show the progress till we get the response from the server.

We should include the `<md-progress-spinner>` at the top of our template; we show it by default and hide it when we get the response:

```
<md-progress-spinner color="accent" mode="indeterminate"
                     [style.display]="spinnerVisibility"
class="spinner">
</md-progress-spinner>
```

In the component, we are using the `spinnerVisibility` property to control the visibility of the `<md-progress-spinner>` component:

```
spinnerVisibility = 'block';

  getBooks() {
    this.bookStoreService
      .getBooks()
```

```
      .subscribe(response => {
        this.booksList = response;
        this.spinnerVisibility = 'none';
      });
  }
```

Here is the output of using the `<md-progress-spinner>` component:
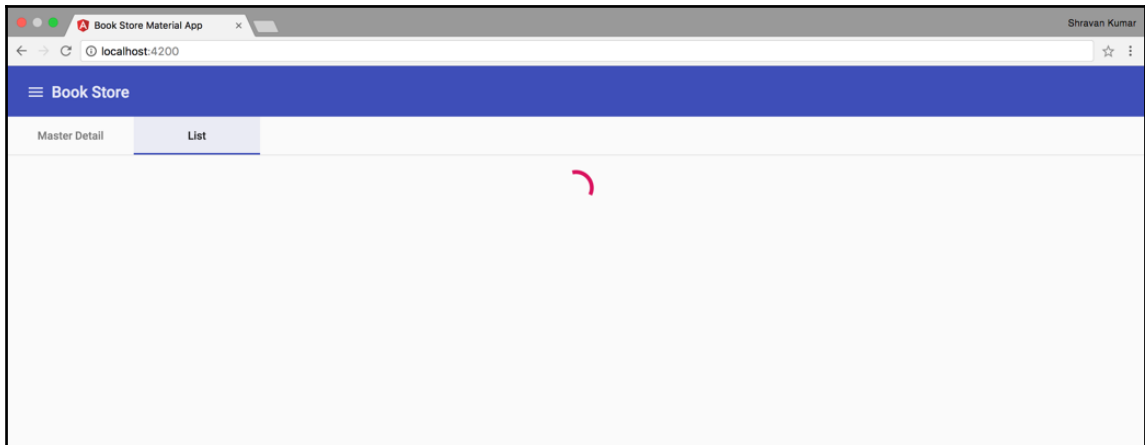
In our examples, we might not be able to see the spinner; to simulate the delay, we use the RxJS `delay()` operator in the `BookStoreService`:

The code for `src/app/books/book-store.service.ts` is as follows:

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

getBooks(): Observable<Book[]> {
  const url = `${this.baseUrl}`;
  return this.http.get(url)
    .delay(5000)
    .map(response => response.json() as Book[]);
}
```

Now our `getBooks()` method waits five seconds to return the response.

We should include `MdProgressSpinnerModule`, `MdSnackBarModule` in `AppMaterialModule`.

Add book dialog

In this section, you are going to learn how to implement a form using Angular Material form controls and a dialog using `MdDialog`:

The code for `src/app/books/add-book-dialog/add-book-dialog.component.ts` is as follows:

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

@Component({
  selector: 'add-book-dialog',
  styleUrls: ['./add-book-dialog.component.scss'],
  templateUrl: './add-book-dialog.component.html'
})
export class AddBookDialogComponent  {
  constructor(private dialogRef:
            MdDialogRef<AddBookDialogComponent>) {}
}
```

In the preceding component, we are injecting the `MdDialogRef` into the constructor, which can be used to refer to the dialog itself.

The code for `src/app/books/add-book-dialog/add-book-dialog.component.html` is as follows:

```
<h3>Add Book</h3>
<form #form="ngForm" (ngSubmit)="dialogRef.close(form.value)"
                                    ngNativeValidate>

  <div fxLayout="column" fxLayoutGap="8px">
    <md-input-container>
      <input mdInput ngModel name="title"
placeholder="Book Title" required>
    </md-input-container>
    <md-input-container>
      <input mdInput ngModel name="authors"
placeholder="Authors" required>
    </md-input-container>
    <md-input-container>
      <input mdInput ngModel name="published"
placeholder="Published" required>
    </md-input-container>
    <md-input-container>
      <input mdInput ngModel name="isbn"
placeholder="ISBN" required>
```

```
    </md-input-container>
    <md-input-container>
      <input mdInput ngModel name="coverImage"
placeholder="Cover Image" required>
    </md-input-container>
    <md-input-container>
      <textarea mdInput ngModel name="description"
placeholder="Description"
rows="3" cols="60" required>
</textarea>
    </md-input-container>
  </div>

  <md-dialog-actions align="end">
    <button md-button type="button"
(click)="dialogRef.close()">Cancel</button>
    <button md-button color="accent">Save Book</button>
  </md-dialog-actions>
</form>
```

In the preceding template, we are using the Angular forms API to bind form controls and the `mdInput` directive with the text input wrapped in `<md-input-container>` for material styles. Finally, we are using `(ngSubmit)="dialogRef.close(form.value)"` to close the form and pass the form value.

We should include `AddBookDialogComponent` to the `declarations` array and `FormsModule` to the `imports` array in `AppModule`.

We can use the earlier component to display the add book dialog. Let us add a button in the toolbar to show the dialog, as follows:

The code for `src/app/app.component.html` is as follows:

```
<md-toolbar color="primary">
    <button md-icon-button (click)="sidenav.toggle()">
      <md-icon>menu</md-icon>
    </button>
    <span>Book Store</span>
    <span class="spacer"></span>
    <button md-mini-fab (click)="openAddBookDialog()">
      <md-icon>add</md-icon>
    </button>
  </md-toolbar>
```

The code for `src/app/app.component.ts` is as follows:

```
import { Component } from '@angular/core';
import { MdDialog, MdSnackBar } from '@angular/material';
import { AddBookDialogComponent, BookStoreService }
                                          from './books';

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

  constructor(private dialog: MdDialog,
 private snackBar: MdSnackBar,
 private bookStoreService: BookStoreService) { }

  openAddBookDialog() {
    this.dialog.open(AddBookDialogComponent)
      .afterClosed()
      .filter(book => !!book)
      .switchMap(book => this.bookStoreService.addBook(book))
      .subscribe(result => {
        if (result.ok) {
          this.openSnackBar();
        }
      });
  }

  openSnackBar() {
    this.snackBar.open('Book Added', 'CLOSE', {
      duration: 1000
    });
  }
}
```

In the component, we are injecting the `MdDialog` into the constructor; whenever the user clicks on the add button in the toolbar, it will invoke the `openAddBookDialog()` method. Inside, we are using the `MdDialog` method's `open()` method to show the add book dialog. Then, we are using `afterClosed()` on `MdDialog` to get the value passed from the `dialogRef.close(form.value)` event in the add book form.

We should include `MdDialogModule` and `MdInputModule` in `AppMaterialModule` and `AddBookDialogComponent` in the `entryComponents` and `declarations` arrays. Dialogs can't be resolved dynamically, so we must add them to `entryComponents`.

Here is the output of `AddBookDialogComponent`:
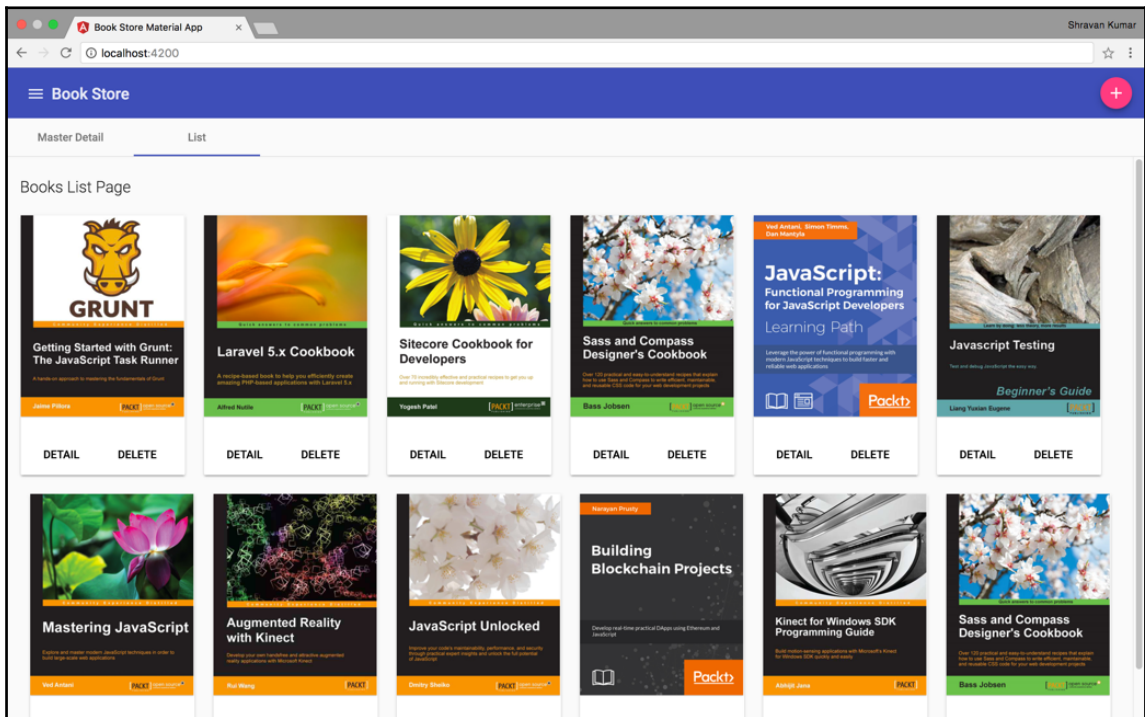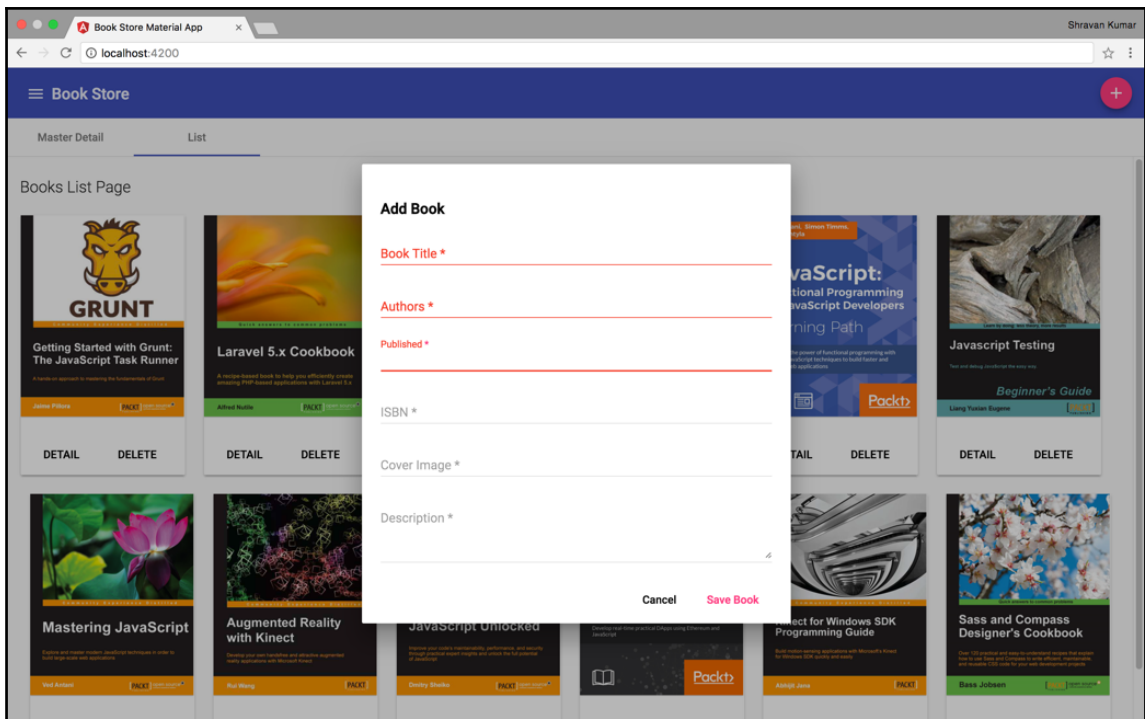
User registration form

To learn the remaining Angular Material controls, let us build the user registration form and create a new application from the current stage of the Book Store application, as follows:

The code for `src/app/user-registration/user-registration.component.ts` is as follows:

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

@Component({
  selector: 'user-registration',
  templateUrl: './user-registration.component.html',
  styles: [`
    .user-registration-form {
      width: 60%;
    }
    .gender-radio-group {
      display: inline-flex;
```

```
      flex-direction: row;
    }
    .gender-radio-button {
     margin: 5px;
    }
  `]
})
export class UserRegistrationComponent {
  countries: Array<object> = [
    {code: 'CA', name: 'Canada'},
    {code: 'SW', name: 'Switzerland'},
    {code: 'IN', name: 'India'},
    {code: 'UK', name: 'United Kingdom'},
    {code: 'US', name: 'Canada'}
  ];

  genders: Array<string> = [
    'Male',
    'Female',
    'Other'
  ];

  submitUserForm(value: Object) {
    console.log(value);
  }
}
```

The code for `src/app/user-registration/user-registration.component.html` is as follows:

```
<md-select name="country" placeholder="Country" ngModel
required>
  <md-option *ngFor="let country of countries"
[value]="country.code">
    {{ country.name }}
  </md-option>
</md-select>
<md-radio-group class="gender-radio-group" ngModel
name="gender" class="m-t" required>
  <md-radio-button class="gender-radio-button"
*ngFor="let gender of genders" [value]="gender">
    {{gender}}
  </md-radio-button>
</md-radio-group>
<md-slide-toggle ngModel name="isAdmin" class="m-t">
Admin User
</md-slide-toggle>
<md-checkbox name="agreement" ngModel class="m-t" required>
```

[194]

```
    I agree to the Terms of Service
  </md-checkbox>
```

The preceding is just a snippet from the `UserRegistrationComponent` template; the code snippet shows the new controls, such as `<md-select>`, `<md-radio-group>`, `<md-slide-toggle>`, `<md-checkbox>`. We should include `MdSelectModule`, `MdRadioModule`, `MdCheckboxModule`, and `MdSlideToggleModule` in `AppMaterialModule` and `UserRegistrationComponent` in the `AppModule` declarations array.

To navigate to the earlier code, we should add routing to our application. Let us refactor our application. Now we need a container component to show the `master-detail` and `books-list` components:

The code for `src/app/books/books-container.component.ts` is as follows:

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

@Component({
  selector: 'books-container',
  template: `
    <md-tab-group>
      <md-tab label="Master Detail">
        <div fxFlex class="master-detail-container">
          <bl-master-detail fxFlex></bl-master-detail>
        </div>
      </md-tab>
      <md-tab label="List">
        <div fxFlex class="books-list-container">
          <bl-list fxFlex></bl-list>
        </div>
      </md-tab>
    </md-tab-group>
  `,
  styles: [`
    .master-detail-container {
      height: calc(100vh - 113px);
      overflow: hidden;
      padding: 1rem;
    }

    .books-list-container {
      height: 100%;
      padding: 1rem;
      overflow-x: hidden;
      overflow-y: auto;
    }
  `]
```

```
})
export class BooksContainerComponent {
}
```

We should also refactor our `AppComponent` to show the components inside of it. In the template, we are adding `<router-outlet>` to show components based on selected routes:

```
<md-sidenav-container fxFlex>
  <md-sidenav mode="over" #sidenav>
    <md-nav-list>
      <a md-list-item [href]="link.path"
              *ngFor="let link of links">
          {{ link.name }}
        </a>
      </md-nav-list>
  </md-sidenav>
  <div class="content" fxLayout="column" fxFlex>
    <router-outlet></router-outlet>
  </div>
</md-sidenav-container>
```

In template, we need to update the `links` array:

```
links = [
    { name: 'Books', path: 'books' },
    { name: 'Registration', path: 'registration' }
];
```

Finally, we need to include the routes in the `AppModule`:

The code for `src/app/app.module.ts` is as follows:

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

const routes: Routes = [
  {path: '', redirectTo: 'books', pathMatch: 'full'},
  {path: 'books', component: BooksContainerComponent },
  {path: 'registration', component: UserRegistrationComponent }
];

@NgModule({
  ...
  imports: [
    ...,
    RouterModule.forRoot(routes)
  ],
```

[196]

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

Here is the output of the user registration form with application routing:

Adding themes

Themes let the user switch between different color schemes. Let us include a `theme.scss` file to our application. Till now, we have used the `indigo-pink` color scheme; let us use the add color scheme to the application:

The code for `src/theme.scss` is as follows:

```
@import '~@angular/material/_theming';

@include mat-core();

$primary: mat-palette($mat-red);
$accent: mat-palette($mat-blue);

$theme: mat-light-theme($primary, $accent);
```

```
@include angular-material-theme($theme);
```

We added the red-blue theme for the application; we can choose any palettes out of the Material design color palettes (`https://material.io/guidelines/style/color.html`).

We also need to include the `theme.scss` file to the `angular-cli.json` file styles array:

```
"styles": [
"styles.scss",
      "theme.scss"
]
```

After adding the `theme.scss` file, we need to restart the application to see the theme effect.

Here is the final output:
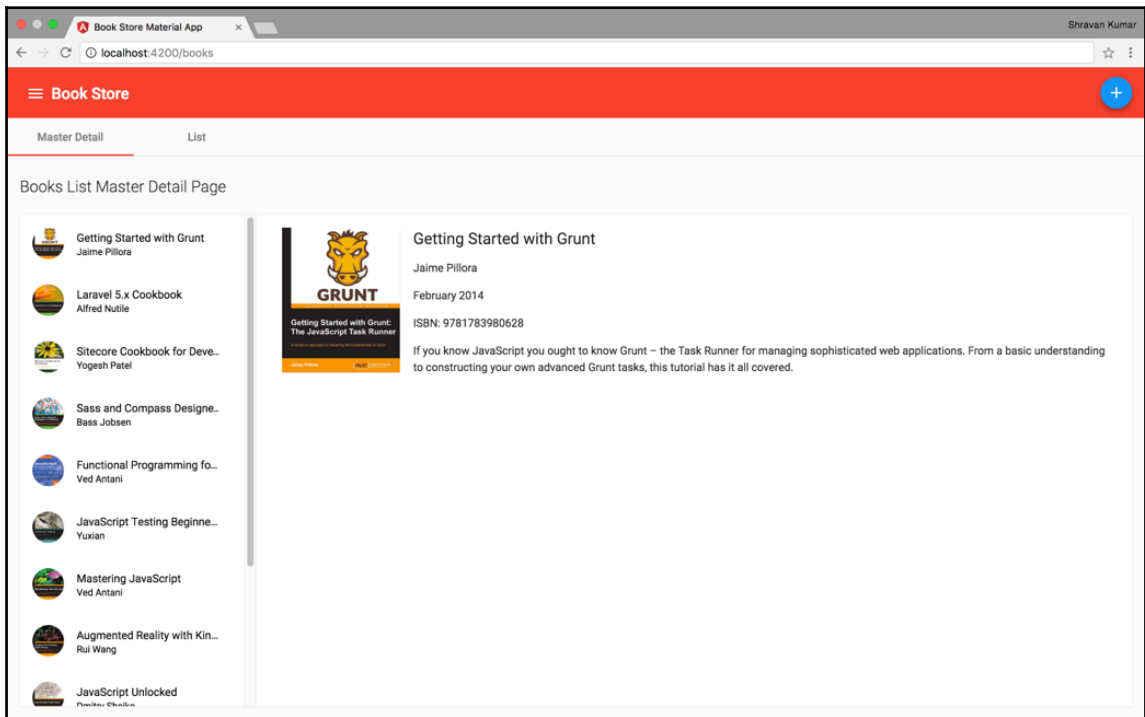
Summary

We started this chapter with the introduction to material design, and we looked at how to add the Angular Material to our project. Then, you learned how to use the flex layout and various Angular Material components to build different kinds of UI, such as the **Master Detail** page, **List** view page, and forms. Finally, we looked at how to add themes from the Material design color palettes.

Index