



This book is provided in digital form with the permission of the rightsholder as part of a Google project to make the world's books discoverable online.

The rightsholder has graciously given you the freedom to download all pages of this book. No additional commercial or other uses have been granted.

Please note that all copyrights remain reserved.

About Google Books

Google's mission is to organize the world's information and to make it universally accessible and useful. Google Books helps readers discover the world's books while helping authors and publishers reach new audiences. You can search through the full text of this book on the web at <http://books.google.com/>

Web Development with Angular and Bootstrap

Third Edition

Embrace responsive web design and build adaptive Angular web applications



Packt

www.packt.com

Sridhar Rao Chivukula and Aki Iskandar

Web Development with Angular and Bootstrap

Third Edition

Embrace responsive web design and build adaptive Angular web applications

**Sridhar Rao Chivukula
Aki Iskandar**

Packt

BIRMINGHAM - MUMBAI

Web Development with Angular and Bootstrap

Third Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Larissa Pinto
Content Development Editor: Keagan Carneiro
Senior Editor: Martin Whittemore
Technical Editor: Jinesh Topiwala
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Designer: Nilesh Mohite

First published: May 2015
Second edition: November 2016
Third edition: August 2019

Production reference: 1300819

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78883-810-8

www.packtpub.com

*To my dear Bhavya, Pratyush, and Aaradhya for bringing joy, happiness, smiles,
and strength into our lives.*

– Sridhar Rao Chivukula



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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.

Contributors

About the authors

Sridhar Rao Chivukula is a technical lead at Mindtree Ltd and is based out of New York City. He brings with him more than a decade of rich hands-on experience in all aspects of frontend engineering. He has worked with leading companies such as Oracle, Tech Mahindra, and Cognizant Technology Solutions. He has a Bachelor's degree in Information Technology. He is the author of the books *Expert Angular* and *PHP and script.aculo.us Web 2.0 Application Interfaces*, published by Packt.

Aki Iskandar is an entrepreneur and software architect with over two decades of programming experience. He has worked as a consultant for Microsoft, Compuware, Progressive Insurance, Allstate Insurance, KeyBank, Ernst & Young, and Charles Schwab, to name a few. His last full-time job, before leaving the corporate world in 2011, was at PNC, where he served as an enterprise architect on their Enterprise Architecture team. During that time, he served as a core member on PNC's Architecture Review Board, which was responsible for creating reference architectures, reviewing the architectural diagrams for IT projects that were in the tens of millions of dollars in size, and establishing IT Governance for the corporation.

He is the founder of VIZCARO.com (an online motivational service that helps thousands of people) and maintains a blog focusing on Angular and related technologies.

About the reviewer

Phodal Huang is a developer, creator, and author. He works at ThoughtWorks as a senior consultant and focuses on IoT and the frontend. He is the author of *Design IoT System*, *Thinking in Full Stack*, and *Frontend Architecture* in Chinese.

He is an open source enthusiast and has created a series of projects on GitHub. After his daily work, he likes to reinvent some wheels for fun. He created the micro-frontend framework Mooa for Angular. You can discover more wheels on his GitHub page, [@phodal1](#).

He loves designing, writing, hacking, and traveling. You can also find out more about him on his personal website at [phodal.com](#).

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](#) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

| | |
|---|----|
| Preface | 1 |
| Chapter 1: Quick Start | 6 |
| Angular's evolution | 8 |
| Angular's building blocks | 9 |
| Components | 10 |
| Templates | 10 |
| Directives | 10 |
| Modules | 10 |
| Services | 11 |
| Dependency injection | 11 |
| Setting up your development environment | 12 |
| Writing your first Angular application | 14 |
| Using your development environment | 14 |
| Location of your files | 15 |
| Generating our to-do list application | 15 |
| Serving up our to do list application | 16 |
| Angular basics | 18 |
| Components | 18 |
| Code listing for the to-do list application | 20 |
| Interpolation | 24 |
| Templating and styling | 26 |
| Property binding | 27 |
| Event binding | 29 |
| Our sample project | 29 |
| Annotated photo album | 30 |
| Design principles | 30 |
| Wireframes | 31 |
| Wireframing tools | 31 |
| Wireframes for our annotated photo album | 31 |
| Home page | 32 |
| Dashboard | 32 |
| Image upload | 33 |
| Photo preparation | 34 |
| Create Album | 35 |
| Photo listing | 35 |
| Photo album listing | 36 |
| Workbench | 37 |
| Album viewer | 37 |
| Paper prototyping | 38 |
| Summary | 40 |

| | |
|---|----|
| Chapter 2: ECMAScript and TypeScript Crash Course | 41 |
| The (quick) roadmap | 42 |
| The relationship between JavaScript and TypeScript | 42 |
| A series of fortunate events for JavaScript | 43 |
| Chromium project | 43 |
| Javascript frameworks | 44 |
| ECMAScript 2015 | 44 |
| TypeScript crash course | 45 |
| Transpilation versus compilation | 45 |
| let | 46 |
| Const | 47 |
| Data typing | 47 |
| Objects | 49 |
| JSON | 50 |
| JavaScript runtime environment | 51 |
| Arrays | 51 |
| TypedArray | 52 |
| Maps | 53 |
| WeakMap | 55 |
| Set | 55 |
| WeakSet | 57 |
| Classes | 59 |
| Interfaces | 61 |
| Inheritance | 63 |
| Destructuring | 64 |
| Template strings | 65 |
| for-of loop | 65 |
| Decorators | 66 |
| Promises | 67 |
| Modules | 68 |
| Default exports | 68 |
| Summary | 69 |
| Chapter 3: Bootstrap - Grid Layout and Components | 71 |
| A word about what this chapter is not | 72 |
| Our example application | 73 |
| Game plan | 74 |
| Sass crash course | 77 |
| What is Sass? | 78 |
| Compass framework | 78 |
| Two SASS styles | 79 |
| Installing Ruby | 80 |
| Installing Sass | 81 |
| Online tools for Sass | 81 |
| Offline tools for Sass | 81 |
| Sass features and syntax | 82 |
| Nesting | 83 |
| Variables | 83 |
| Mathematical operations | 84 |
| Imports | 85 |
| Extend | 86 |

Table of Contents

| | |
|---|-----|
| Mixins | 87 |
| Placeholders | 90 |
| Built-in functions | 90 |
| Custom functions | 91 |
| Bootstrap crash course | 93 |
| What is Bootstrap? | 93 |
| Motivation | 94 |
| Bootstrap's role in our example application | 95 |
| Installing Bootstrap | 95 |
| Bootstrap's responsive grid system | 96 |
| The container | 98 |
| The row | 99 |
| The column | 102 |
| Differing viewport sizes | 103 |
| Bootstrap components | 105 |
| Button components | 106 |
| Alert components | 108 |
| Navbar component | 109 |
| Modal components | 109 |
| Listing Carousel – a formal introduction | 110 |
| Idea generation/concept | 111 |
| Analysis — feasibility study | 112 |
| Requirement gathering | 113 |
| Wireframes | 114 |
| Implementation | 115 |
| Installing our interim web server | 115 |
| Welcome page | 115 |
| Signup | 120 |
| Login | 122 |
| Listings | 124 |
| Create listing | 127 |
| Edit listing | 127 |
| Wireframes collection | 129 |
| Summary | 135 |
| Chapter 4: Routing | 136 |
| What is routing in Angular? | 137 |
| Creating our application's shell using the CLI | 138 |
| First things first – basic concepts | 143 |
| Base Href | 143 |
| RouterLink and RouterLinkActive | 144 |
| Configuring routes for our application | 145 |
| Parameterized routes | 147 |
| Child routes | 148 |
| Route guards | 149 |
| Completing our route configuration | 152 |
| Bootstrap navigation bar and router links directives | 154 |
| Specifying the location for rendering the component templates | 157 |
| Running our application | 157 |
| Routing strategies | 158 |

| | |
|--|-----|
| Summary | 160 |
| Chapter 5: Flex-Layout - Angular's Responsive Layout Engine | 161 |
| Why this chapter was included in the book | 162 |
| The four available techniques for the layout of our components | 162 |
| Table | 164 |
| Positioning using float, and clear | 164 |
| FlexBox CSS | 166 |
| CSS Grid | 166 |
| Why FlexBox CSS is probably the best option | 166 |
| What is Flex-Layout and why should you use it? | 167 |
| Integrating Flex-Layout | 168 |
| The Flex-Layout API | 170 |
| Design strategies when using FlexBox | 171 |
| Associating our components with the chapters and topics in this book | 172 |
| Implementing our selected wireframes | 174 |
| The Create Listing wireframe | 176 |
| The Edit Listing wireframe | 179 |
| Summary | 182 |
| Chapter 6: Building Angular Components | 183 |
| Angular application architecture – a tree of components | 184 |
| Architecting an Angular application | 184 |
| Breaking up your components into sub-components | 185 |
| Component responsibilities | 186 |
| Annotations | 186 |
| @Component | 186 |
| Properties of the @Component decorator | 187 |
| selector | 187 |
| template and templateUrl | 187 |
| styles and stylesUrls | 188 |
| View encapsulation | 189 |
| Module versus NgModule | 190 |
| Properties of the @NgModule decorator | 190 |
| Content projection | 191 |
| Projecting multiple sections | 192 |
| Life cycle hooks | 193 |
| Most common life cycle hooks | 193 |
| Component interface – inputs and outputs, and the flow of data | 194 |
| Our implementation of the components for our three pages | 196 |
| Summary | 197 |
| Chapter 7: Templates, Directives, and Pipes | 199 |
| Templates | 199 |
| Directives | 200 |
| Attribute directives | 201 |
| Structural directives | 202 |
| NgFor | 202 |
| Accessing the index value of the iteration | 203 |

Table of Contents

| | |
|--|-----|
| Built-in directives | 204 |
| NgIf | 204 |
| NgSwitch, NgCase, and NgDefault | 205 |
| NgStyle | 207 |
| NgClass | 209 |
| NgNonBindable | 210 |
| Data binding using the NgModel directive | 211 |
| Event binding | 212 |
| Property binding | 212 |
| Custom directives | 212 |
| Pipes | 213 |
| Custom pipes | 216 |
| Summary | 216 |
| Chapter 8: Working with NG Bootstrap | 218 |
| Integrating NGB | 219 |
| Installing NGB | 219 |
| Why use NGB? | 222 |
| Creating our playground for NGB (and Angular Material, and more) | 223 |
| Creating a playground directory | 224 |
| Creating the playground component class | 224 |
| Creating the playground template file | 225 |
| Creating the playground CSS file | 225 |
| Creating the playground menu item | 226 |
| NGB widgets | 226 |
| Collapse | 227 |
| Our parent component | 227 |
| Our NGB collapse component class | 228 |
| Our NGB collapse component template | 229 |
| Importations and declarations | 230 |
| Modal | 230 |
| Our NGB modal component class | 231 |
| Our NGB modal component template | 232 |
| Importations and declarations | 233 |
| Carousel | 234 |
| Our NGB carousel component class | 234 |
| Our NGB carousel component template | 235 |
| Implementing NGB into our example application | 236 |
| UX design rules of thumb | 238 |
| Keep it clean | 238 |
| Keep it functional | 239 |
| Keep it obvious | 239 |
| Summary | 240 |
| Chapter 9: Working with Angular Material | 241 |
| What is Angular Material? | 242 |
| Installing Angular Material | 243 |
| Categories of components | 247 |

| | |
|--|-----|
| Navigation | 250 |
| Navigation components using schematics | 250 |
| Custom Material menus and navigation | 251 |
| Custom sidebar menus | 253 |
| Cards and layout | 254 |
| Material cards | 255 |
| Lists | 256 |
| Lists with dividers | 257 |
| Navigation lists | 257 |
| Accordions and expansion panels | 258 |
| Steppers | 260 |
| Tabs | 261 |
| Form controls | 261 |
| Buttons and indicators | 265 |
| Popups and modals | 270 |
| Data tables | 273 |
| Summary | 274 |
| Chapter 10: Working with Forms | 276 |
| Bootstrap forms | 276 |
| What are forms? | 277 |
| Bootstrap form classes | 277 |
| Bootstrap form classes – extended | 280 |
| Sizing | 281 |
| Readonly | 282 |
| Inline forms | 283 |
| Forms using Bootstrap grid classes | 284 |
| Disabled | 285 |
| Help text inside forms | 287 |
| Displaying input elements as plain text | 288 |
| Angular forms | 289 |
| Template-driven forms | 290 |
| Template-driven forms – pros | 291 |
| Template-driven forms – cons | 291 |
| Template-driven forms – important modules | 291 |
| Building our login form | 292 |
| Model-driven forms, or reactive forms | 295 |
| Model-driven forms – pros | 296 |
| Model-driven forms – cons | 296 |
| Model-driven forms – important modules | 297 |
| Reactive forms – registration form example | 298 |
| Angular form validations | 301 |
| Template-driven form validation | 302 |
| Reactive form, or model-driven form, validations | 304 |
| Submitting form data | 306 |
| Summary | 308 |
| Chapter 11: Dependency Injection and Services | 309 |

| | |
|--|-----|
| What is DI? | 310 |
| Generating services and interfaces | 314 |
| Guarding against code minification | 317 |
| Summary | 317 |
| Chapter 12: Integrating Backend Data Services | 318 |
| ListingApp – an overview | 319 |
| Fundamental concepts for Angular applications | 320 |
| Strongly typed languages | 321 |
| TypeScript interfaces | 321 |
| Observables | 322 |
| NoSQL databases concept | 322 |
| CRUD operations – overview | 323 |
| ListingApp – technical requirements | 324 |
| Building APIs for ListingApp | 324 |
| Google Firestore database | 328 |
| Angular HttpClient | 331 |
| HttpClient and HTTP verbs | 332 |
| HTTP GET | 333 |
| HTTP POST | 334 |
| HTTP PUT | 334 |
| HTTP DELETE | 335 |
| HTTP via promises | 335 |
| Integrating backend services | 337 |
| Integrating Angular HTTP with backend APIs | 338 |
| Integrating Angular HTTP with Google Firebase | 351 |
| Summary | 358 |
| Chapter 13: Unit Testing | 359 |
| Introduction to testing frameworks | 359 |
| About the Jasmine framework | 360 |
| About the Karma framework | 361 |
| Angular test automation | 362 |
| Testing Angular components | 362 |
| Testing directives | 367 |
| Testing Angular routing | 374 |
| Testing dependency injection | 378 |
| What is dependency injection? | 378 |
| Testing Angular services | 378 |
| Testing HTTP | 384 |
| Summary | 389 |
| Chapter 14: Advanced Angular Topics | 390 |
| Custom directives | 390 |
| Custom form validations | 393 |
| Building single-page applications | 396 |

Table of Contents

| | |
|--|-----|
| User authentication | 397 |
| User authentication with Firebase | 398 |
| User authentication with Auth0 | 407 |
| Summary | 418 |
| Chapter 15: Deploying Angular Applications | 419 |
| Deploying Angular applications | 419 |
| Compilation options for Angular applications | 420 |
| What is just-in-time compilation? | 420 |
| What is ahead-of-time compilation? | 421 |
| Deploying a standalone Angular application | 421 |
| Deploying composite Angular applications | 427 |
| Creating and deploying multiple Angular applications | 428 |
| Packing the Angular project as an npm package | 440 |
| Deploying Angular apps to GitHub Pages | 443 |
| Creating and deploying applications in GitHub Pages | 444 |
| Summary | 451 |
| Other Books You May Enjoy | 452 |
| Index | 455 |

Preface

Modern web application development has evolved a lot in recent years. The applications are designed and developed using the mobile-first approach. Angular is every developer's dream framework that is used for rapid prototyping in order to deploy complex enterprise applications. This book will guide you in jump-starting Angular application development using the Bootstrap CSS framework. Each chapter in this book is structured to give you a full understanding of a specific topic and aspect of the Angular framework, along with other related libraries and frameworks. Each of these aspects are essential for designing and developing a robust Angular application. In a step-by-step approach, you will learn about each of the following phases: TypeScript, Bootstrap framework, Angular routing, components, templates, Angular Material, dependency injection, and much more.

By the end of the book, you will be able to create modern, responsive, cross-platform Angular applications using various CSS frameworks, including Bootstrap, Material Design, and NG Bootstrap.

Who this book is for

This book has been carefully thought through, and its content covers topics ranging from those suitable for absolute beginners to some of the most complex use cases. The detailed explanations, step-by-step hands-on use case examples, and easy flow of this book make it a must-have for your collection.

This book will benefit developers with little or no programming background, while also benefiting seasoned developers.

What this book covers

Chapter 1, *Quick Start*, begins your journey with the help of a quick introductory chapter to show you the possibilities available and to get you thinking creatively.

Chapter 2, *ECMAScript and TypeScript Crash Course*, examines the TypeScript language, from basic to advanced aspects of the language that are essential when it comes to writing Angular applications.

Chapter 3, *Bootstrap – Grid Layout and Components*, introduces the awesome Bootstrap CSS framework and explains how you can use some of the components and utilities provided by this supersonic library.

Chapter 4, *Routing*, enables you to cut your teeth on the Angular framework by learning all about routing. From defining simple route paths to complex route guards and much more, you will be empowered with the knowledge to build rock solid routing systems for your applications.

Chapter 5, *Flex Layout – Angular's Responsive Layout Engine*, covers alternative layouts and a grid-designing library named Flex Layout, and explains how Flex Layout offers a powerful layout for your Angular applications.

Chapter 6, *Building Angular Components*, covers Angular components, the main building blocks of modern progressive web applications. You will learn to build multiple components and bring them all together to build views and functionality.

Chapter 7, *Templates, Directives, and Pipes*, introduces the Angular template engine, directives, and pipes. You will explore the built-in directives, pipes, and templates.

Chapter 8, *Working with NG Bootstrap*, introduces NG Bootstrap, another super powerful framework that you can consider in your projects.

Chapter 9, *Working with Angular Material*, explains the development of our Angular applications using the components, directives, and utilities provided by Angular Material.

Chapter 10, *Working with Forms*, introduces the heartbeat of any dynamic application. You will learn about the different approaches to building forms, exploring template-driven forms, reactive forms, and much more.

Chapter 11, *Dependency Injection and Services*, covers dependency injection, services, and the design philosophy behind the scenes.

Chapter 12, *Integrating Backend Data Services*, is where all the code, learning, and hands-on examples that you have learned and built throughout the chapter will come together. You will stitch end-to-end screens, from UIs to components, to services and models, and much more. You will learn everything you need in order to assimilate all aspects of Angular.

Chapter 13, *Unit Testing*, explains how testing is one of the most important aspects in modern-day software development. This chapter will teach you all about testing Angular applications.

Chapter 14, *Advanced Angular Topics*, discusses user authentication and powerful user management systems, and also covers the integration of your applications with Google Firebase and Auth0.

Chapter 15, *Deploying Angular Applications*, examines how to deploy your Angular applications and make them production ready.

To get the most out of this book

To make the most of the information that you will learn in this book, you are encouraged to do a quick recap of programming fundamentals, including classes, objects, loops, and variables. This can be in any language of your choosing. Angular applications are not limited to any particular operating system, so all that is required is a decent code editor and a browser. Throughout the book, we have used Visual Studio Code editor, which is an open source editor and is free to download.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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 code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Web-Development-with-Angular-and-Bootstrap-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The screenshot shows the edited `angular.json` file."

A block of code is set as follows:

```
"styles": [
    "styles.css",
    "./node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
    ".../node_modules/jquery/dist/jquery.min.js",
    ".../node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AppComponent } from './app.component';
import { JacketListComponent } from '../../../../../projects/jackets/src/app/jacket-list/jacket-list.component';
import { VendorsComponent } from
'../../../projects/vendors/src/lib/vendors.component';
```

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

```
ng new realtycarousel
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "We have added the menu link, **Add New Listing**."

Warnings or important notes appear like this.



Tips and tricks appear like this.



Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1 Quick Start

Are you ready to work your way to Angular mastery? My guess is that you are, and with this book and your determination, you will. You purchased this book, and I not only thank you, but I want to make a promise to you right here, right now. Two promises, in fact. The first one is that if you diligently read the material, apply the knowledge you'll gain along the way, and build the example application with me throughout these pages, you will be well on your way to Angular mastery.

If you're anything like me, you have a library packed with literally hundreds of technical books and you have read most of them. Some books start off at an excruciatingly slow pace, entrenched in theory and historical minutia, while other books start off so quickly, they leave the readers scratching their heads wondering if they are too dumb to understand the material. The truth is, striking a balance while introducing readers to potentially brand new material—and not have them nod off as they read their newly acquired 400+ page technical book—is a tricky thing to pull off. So, my esteemed budding Angular guru, that is what my second promise is to you. I promise to do my utmost in finding that all-elusive happy medium between being technically heavy-handed, and being real-world practical while making this book as entertaining a read as I possibly can for you.

With the promises well stated, let's start our journey to Angular mastery together by taking a quick look at what we're going to cover in this brisk, but all important first chapter.

We'll set up your development environment in a jiffy, and we'll build our first Angular application in order to get some immediate feeling of accomplishment. We're going to skim over the details as we write it, but right after that, we'll cover a few critical Angular basics in more detail before concluding this chapter. These first few basics are the very first things you should get comfortable with because we'll use them over and over again throughout the book as we learn about the more advanced stuff.

After we cover these basics, we'll switch gears from Angular speak and we'll take a look at the fully fledged application that we'll be building together throughout the remainder of the book. As a bonus (and there are a few bonuses in this book that I hope will bring you a bunch of value), we're also going to touch on design principles, wireframing, and a little used design strategy called paper prototyping—whose focus is on usability testing. Paper prototyping has been around since approximately 1985 and has largely been replaced by Lean UX design since approximately 2008. Still, I'm always amazed at how many of my clients have never even heard of paper prototyping—but I'm also happy when they discover the value it brings to them when they try it out.

We'll cover paper prototyping at a high level at the end of this chapter, immediately following the wireframes section, which is the most logical place for discussing paper prototyping. We'll touch on a few UX design principles as well, but not on the Lean UX process itself since that would take us too far off course from our focus of this book. However, if you have an interest in the Lean UX design process, here is a good starting point: <https://www.interaction-design.org/literature/article/a-simple-introduction-to-lean-ux>.

OK, my esteemed budding Angular guru, are you primed and ready to get started? Good! Let's go!

The topics we will be covering in the chapter are:

- Angular's evolution
- Angular's building blocks
- Setting up your development environment
- Writing your first Angular application
- Angular basics
- Our sample project
- The process of paper prototyping

Angular's evolution

Angular is a frontend JavaScript-based web application framework that provides you with everything you need, including the kitchen sink, with which to build amazingly powerful **Single Page Applications (SPAs)**. The application we'll be building together is an SPA, and we'll be discussing SPA strategies along the way.

While Angular wasn't the first JavaScript-based frontend web application framework, it was quite possibly the most powerful one of them. This is likely due to Angular's SPA-centric focus—since there's more to building an SPA application than there is to simply provide two-way data binding on your web pages.

The initial release of Angular was in late fall of 2010. Since then, dozens of competing libraries and frameworks have come on the scene, including some that also have large adoptions and large production implementations, such as Ember.js, Backbone.js, and React.js. Angular, despite having perhaps the highest learning curve (and we'll see why this is the case), remains the most powerful one of them all.

The Angular naming and versioning jungle can, at first glance, seem confusing. There are a few reasons for this, as follows:

- **Angular's 1.x releases:** Basically, any release prior to Angular 2 is commonly referred to as AngularJS.
- AngularJS is no more actively in development mode. It has been put under long term support mode.
- Angular framework is being actively developed, and so developers need to be specific about which of the two Angular frameworks they are referring to when discussing them. Fortunately, there are two completely dedicated websites for them: <https://angularjs.org/> and <https://angular.io>, respectively. The Angular team adopted semantic versioning, starting with the 2.0.0 release. You can read more about semantic versioning here: <https://semver.org>.
- Angular 2 was a complete rewrite of Angular 1.x (that is, AngularJS), and is thus not backward compatible with AngularJS. While Angular 4 was not a complete rewrite of Angular 2, it did have some changes in its core libraries that required the team to increment its major number from 2 to 4. Version 3 was skipped as a release number altogether.

- All releases from Angular 2 onward are commonly referred to as Angular 2+, or simply just as Angular.
- Due to having adopted semantic versioning, the Angular team never released Angular 3 and so went straight from Angular 2 to Angular 4. Specifically, there was a misalignment of the router package's version, which was already distributed as version 3.3.0. We'll cover routing in Angular in great detail in Chapter 4, *Routing*. Don't let this confuse you. Just know that there wasn't ever an Angular 3. No big deal. In the Windows OS world, there was never a Windows 9 either. These things happen.

After reading the preceding list, you can see why there tends to be some confusion around Angular. However, it's pretty straightforward if you keep these two things in mind:

- You should really only be using Angular, and not AngularJS (unless you have an exceptionally good reason for it)
- With the exception of there not being an Angular 3, there will be two major releases per year; they should be contiguous in numbering scheme (that is, 8, 9, and so on), and they are expected to be backward compatible—at least within the same major version number (as per the spirit of semantic versioning)

You can check out the official Angular release schedule here: https://github.com/angular/angular/blob/master/docs/RELEASE_SCHEDULE.md. Since Angular is a complete rewrite of the AngularJS platform, and this is worlds apart from AngularJS, we'll skip AngularJS altogether and start by taking a look at Components which are Angular's building blocks. Are you with me? Great, let's speedily move ahead.

Angular's building blocks

Adding new features is the business of publishing new frameworks—but luckily, the fundamental underlying architecture does not change very often. When it does, it's not typically a complete overhaul. With the exception of Angular 2.0, which was completely divergent from its predecessor, all major releases so far contain largely the same architecture.

Let's now take a look at the core architectural pieces of the framework.

Components

Components are like widgets that are in charge of displaying themselves along with the data they consume and/or manipulate on areas of your screen called views. An Angular application is like a tree of components, and Angular provides mechanisms for components to communicate with each other in a bidirectional manner—parent to child and child to parent.

Templates

Components rely on their templates for rendering their data. Templates are where you define what the component looks like and you can hook in styles to window-dress your component any way you like. A component can either contain its template (that is, the HTML) and its styling (that is, the CSS) either directly within itself or have references to template and style files outside of itself. At the end of the day, the world's fanciest frontend frameworks produce HTML, CSS, and JavaScript because these three things are the only things browsers understand.

Directives

Within the templates you create for your component, Angular enables you to alter the DOM with powerful constructs called directives. There are directives for controlling the way things are rendered on the screen (that is, the component view) such as repeating snippets of HTML, for displaying things based on conditional logic, for hiding or showing things, filtering arrays of data, and much more.

Modules

Angular is modular. That is to say that its functionality is wrapped up in modules, known as NgModules, and are themselves libraries. Modules are perfect for lumping code together in an organized way. For instance, there are modules for helping with forms, routing, and communicating with RESTful APIs. Many third-party libraries are packaged as NgModules so you can incorporate them into your Angular applications. Two examples of this are Material Design and AngularFire—we'll be taking a look at both of these libraries in later chapters.

Services

Services are not really an Angular artifact per se, but rather a very general notion representing encompassed functionality, functions, and features that your application's components may need to consume. Things such as logging, data retrieval, or virtually any calculation or lookup service, can be written as services—these services can reside within your application, or live externally from it. You can think of a service as a highly specialized class that provides some *service* (such as looking up the distance between two zip codes) and does it well. Just as with components, not only are there tons of third-party services you can use in your Angular applications, but you can create your own custom services. We'll learn how to do this in Chapter 12, *Integrating Backend Data Services*.

Dependency injection

Dependency injection(DI), or **Inversion of Control (IoC)**, is a very useful and common software design pattern. This pattern is used to *inject* objects into the objects that depend on them. The object you're using that depends on other objects can just go ahead and use it without needing to worry where it is in order to load it, or how to instantiate it—you just use it as if it just sprung into existence at the time you needed it. Services are perfect for injecting into our application. We'll learn how to use DI in Angular, and how we can use the Angular **command-line interface (CLI)** to generate injectable services of our own design.

Just before we move on to setting up our development environment, here are some interesting things about Angular:

- AngularJS was built using JavaScript, while Angular was built using TypeScript. While this adds a level of abstraction when writing Angular applications, using TypeScript provides a few important advantages when building larger apps with larger teams—we'll get to those shortly.
- AngularJS was based on controllers, whereas Angular is component based. You'll learn all you need to know about components in Chapter 6, *Building Angular Components*.

- SPAs are notorious for being difficult for implementing **Search Engine Optimization (SEO)**, but Angular is SEO friendly.
- It's possible to build native mobile applications with Angular.
- It's also possible to build cross-platform, desktop-installed applications with Angular.
- Angular can also run on the server, using Angular Universal.

You have to admit, this is a pretty impressive and exciting list. These things and more make learning Angular a worthwhile endeavor and the marketplace is asking for Angular know-how.

Setting up your development environment

In order to get started with Angular, you're going to need to have the **Angular CLI** installed; to install that, you first need to have Node.js and **npm (node package manager)** installed. If you've already got Node.js and npm installed, great! If not, don't worry—they are easy to install and I will take you through the installation process in Appendix A, *Toolchain for Web Development with Angular*, near the back of the book. In Appendix A, I also take you through installing the Angular CLI and how to use it for building Angular applications. For brevity, I'll refer to the Angular CLI tool as just the CLI from this point forward.

If you are unsure whether you have NodeJS and npm installed, you can check really quickly by typing `$ node -v` and `$ npm -v`, respectively, on your command line. Similarly, you can type `$ ng -v` on the command line to see whether you have CLI installed. If you get a version number back, you have that particular tool installed (as shown in the following screenshot I took).



Note: Do not type \$ at the beginning of the commands. The \$ signifies the command prompt, entry point for the commands you'll type. Unix-based operating systems, such as macOS and Linux systems, commonly use \$ or % as their command prompt—depending on the shell being used, or if there are any custom settings specified in a configuration file on your system. Windows operating systems typically use the greater than sign, >, as their command prompt.

A screenshot of a terminal window on a dark background. At the top, it displays the Angular CLI logo in red and yellow. Below the logo, the text "Angular CLI: 8.3.1" is followed by "Node: 12.9.1", "OS: win32 x64", and "Angular:" with three ellipses. A table then lists installed packages with their versions:

| Package | Version |
|----------------------------|---------|
| @angular-devkit/architect | 0.803.1 |
| @angular-devkit/core | 8.3.1 |
| @angular-devkit/schematics | 8.3.1 |
| @angular/cli | 8.3.1 |
| @schematics/angular | 8.3.1 |
| @schematics/update | 0.803.1 |
| rxjs | 6.4.0 |

If any of these commands go unrecognized, jump on over to Appendix A real quick, install the tools, and jump right back here. I'll be waiting for you.

We'll also need a code editor. There are many code editors available today, including a number of free ones. Though any code editor will suffice, I would suggest you use Visual Studio Code for your Angular development—at least while working through this book. The reason for this is that Visual Studio Code is free, it's cross-platform, and is an excellent code editor. It's also the code editor that I've used while writing this book and so when I suggest the use of an extension, you can easily install the same one.

The preceding is all you need for this first chapter. When we start building the example project, which requires us to have a local database, you'll also need to install MongoDB. MongoDB, also known as Mongo, is a great NoSQL database that is also free and cross-platform. I take you through Mongo's installation process in Appendix B, *MongoDB*.

Additionally, there will be other software that you'll need to install, such as Chrome extensions, and I will let you know what they are and where to find them at the appropriate time. For now, let's get started with writing some Angular code.

Writing your first Angular application

When it comes to experimenting with Angular code, as you pick up this awesomely powerful framework, you generally have two choices of how to proceed. The first is to use an online code editor such as JSFiddle, Plunker, StackBlitz, or more. In Appendix C, *Working with StackBlitz*, you'll learn about the basics of using StackBlitz so you can use it from time to time to test some quick code without needing a test project in your development environment. You can visit the StackBlitz website here: <https://stackblitz.com>.

The second approach is to use your own local development environment—and since we've already set it up in the previous section, you can create a project whose sole purpose is to run some quick example code if you'd rather use that than an online code editor. My goal is to show you that you have options—there's not just one way to experiment with some code as you learn Angular.

When you use an online code editor, such as StackBlitz, the only software you need installed is a browser—no other tools whatsoever. While this makes things very easy, the trade-off is that you are extremely restricted in what you can do. That being said, I encourage you to experiment with an online code editor, but we'll only be using our development environments throughout this book. So, let's do that and create a small application together in just a few minutes time—we'll build a to-do list app.

Using your development environment

From this point forward, we'll be using our terminals, the CLI, and the Visual Studio Code. Head on over to <https://code.visualstudio.com>, where you'll be able to download the Visual Studio Code installation package for your operating system of choice.

Location of your files

When it comes to setting up a local environment, you can, of course, place your directories and files wherever you like. If you have a folder where you have your web application projects, go to it now. If you don't have a dedicated place for projects, this is as good a time as any to get into the habit of being organized. For instance, on my machine, I have a folder named `dev` for any and all development I do. Within my `dev` folder, I have a folder named `playground`, where I have a subfolder for each technology I'm learning, or playing with. I prefer to use a Mac when writing code, and so my complete pathname to where I have my Angular *play stuff* is `/Users/akii/dev/playground/angular` (as shown at the bottom of the screenshot of my terminal, a few pages back). That same screenshot also shows the versions of Node.js, npm, and the CLI that I had installed at the time of writing. If having a directory structure like this works for you, by all means, use it. If you already have a way you organize your work, use that. The important thing is to be very disciplined and consistent with how you organize your development environment.

Generating our to-do list application

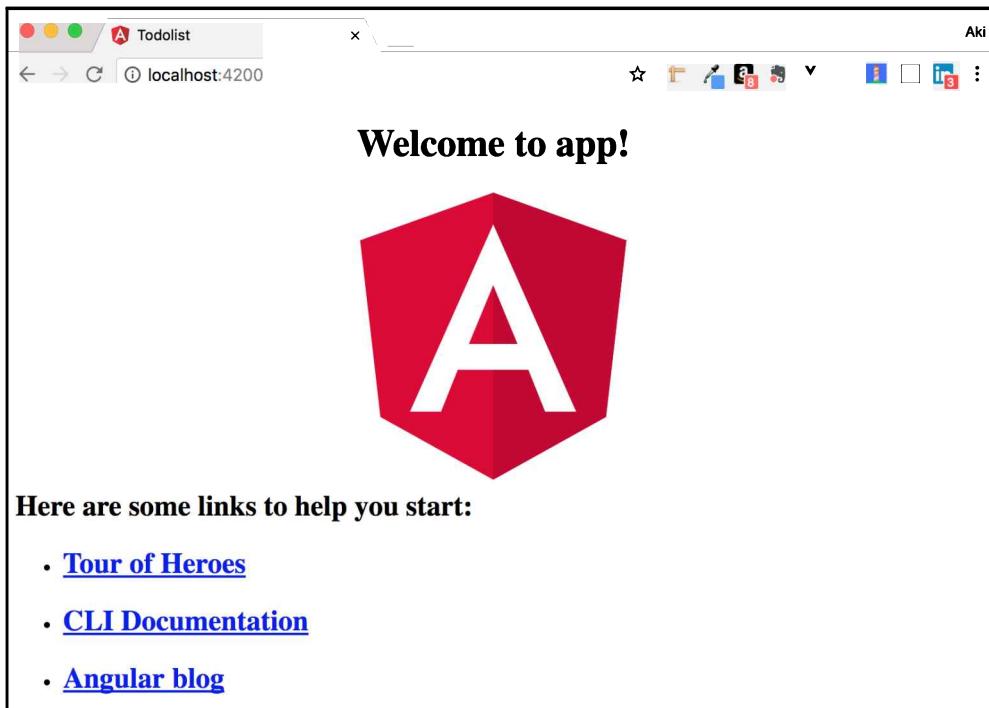
With the installation of what we need now being out of the way—meaning we have our CLI tool available to us—go to your terminal and type the following at your command prompt `$ ng new to-dolist --style=scss --routing`, and then hit *Enter*.

The `ng` command runs the CLI tool, and its `new` command instructs it to create a new Angular application. The application, in this case, is named `to-dolist`. You'll notice that there also are two command-line flags, which are special options for the `new` command. The `style` flag tells the CLI that we'd like to use `scss`, instead of `css`, and the `routing` flag tells the CLI that we'd like it to integrate and set up routing for us by default. We'll be using SASS, as opposed to CSS, in this book, and SCSS is the file extension for Sass files. As a reminder, we're going to have a crash course on Sass in Chapter 3, *Bootstrap – Grid Layout and Components*.

The first time you use the CLI to create your Angular application, it will take anywhere from 45 seconds to well over a minute for it to create your project for you. This is because it needs to download and install various things for you before it creates the project's folder structure. However, when creating subsequent Angular applications, the CLI won't take nearly as long.

Serving up our to do list application

Once the CLI has completed creating the application, you can run it by changing to the project directory (`$ cd to-dolist`) and issuing the `$ ng serve` command. This will have the CLI's built-in web server will be listening on localhost port 4200 by default. By the way, the CLI's web server keeps watch on your project files, and when it notices a change in one of your files, it reloads the application—there's no need for you to stop the server and issue the server command again. This is very convenient as you're making lots of changes and tweaks during development. Next, open your browser and visit `http://localhost:4200` and you should see something like the following, which proves that the CLI is working correctly:



- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Now that the CLI created the to-do list application for you, open that folder in Visual Studio Code (note: for brevity, I'll be referring to Visual Studio Code as the IDE). You should see a folder structure for your to-do list project in the left-hand panel of your IDE, similar to the following (except for the to-do folder, which you won't have just yet; we'll get to how to generate that using the CLI in the upcoming subsection on components).

The following is a screenshot of the **to-dolist** project in the IDE (with the `app.component.ts` file open):

The screenshot shows the Visual Studio Code interface with the 'todolist' project open. The Explorer sidebar on the left displays the project structure:

- e2e
- src
 - app
 - todo
 - app-routing.module.ts
 - app.component.html
 - app.component.scss
 - app.component.spec.ts
 - app.component.ts
 - app.module.ts
- assets
- environments
- favicon.ico
- index.html
- main.ts
- polyfills.ts
- styles.scss
- test.ts
- tsconfig.app.json

The 'app.component.ts' file is currently selected and shown in the main editor area. The code content is:

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

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

The status bar at the bottom indicates the file is on 'master*' branch, has 0 changes, 1 warning, and is using TypeScript 2.6.2.

When working on your Angular applications, the vast majority of your time will be spent working within the `src | app` folder.

Angular basics

Components are the basic building blocks of Angular. In fact, you can think of an Angular web application as a tree of components. When you use the CLI to generate the shell for your Angular application, the CLI also autogenerated one component for you. The filename is `app.component.ts` and is in the `src/app` folder. The `app` component is how your Angular application is bootstrapped—meaning that it is the first component that is loaded, and all the other components are pulled into it. This also means that components can be nested. The previous screenshot shows our project directory structure, with the `src/app` folder expanded, and the `app.component.ts` open in the IDE's file editor window. The `.ts` file extension indicates that it is a TypeScript file. Interesting note—when you write Angular applications, you use TypeScript instead of JavaScript. In fact, the Angular team used TypeScript to write Angular!

After the following *Components* section, you'll find a complete code listing for our Angular application. There are six files that you'll have to edit. Three of them are already available in the application that you generated with the CLI. The other three will be available in your project once you use the CLI to generate the to-do component, which is the directory that you are presently missing when you compare your project structure to the previous screenshot. You'll find out how to do that in the following *Components* section, which is why the complete code listing was inserted afterward. Don't worry—follow along, keep the faith that Angular mastery is within your grasp, and you'll be just fine. If you don't believe me, just lie down on the floor and mumble these words, *This too, shall pass*, three times, slowly.

Components

This section is a high-level fly-by on Angular components—just enough coverage of what an Angular component is. Chapter 6, *Building Angular Components*, is completely dedicated to Angular components and is where we're going to take a deep dive into them. Consider this section to be a little peek behind the component curtain, and when we get to discussing components, we're going to pull the curtains wide open and take a good look at the *Component Wizard of Oz*. Remember that in the *Wizard of Oz* story, Dorothy and the gang were petrified of the Wizard, but when he was finally revealed behind the curtains, they all soon stopped being scared.

As previously mentioned, you can think of components as the basic building blocks or Angular, and of your Angular application as a tree of nested components. Buttons, progress bars, input fields, entire tables, advanced things such as carousels, and even custom video players—these are all components. The components on your web page can communicate with each other, and Angular has a couple of rules and protocols for how they can go about doing so. By the end of this book, you will become very comfortable with the ins and outs of components. You must, for it's simply the way of the Angular guru!

When you write a component file, as in the code that follows, there are three main sections to it. The first is the import section. The middle section is the component decorator, and it's where you indicate what the component's template file is (which defines what the component looks like), and what the component's style file is (which is used to style the component).



Note: Since we used the `style=scss` flag, we get our file in SCSS as opposed to the traditional CSS type file. The export section is the last section in the component file and is where all the logic for the component will be placed. There's a lot more that can go into a component's TypeScript file than what is shown in the following code snippet, as we'll see in Chapter 6, *Building Angular Components*.

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

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

The CLI created the app component for us by default when it created our application for us, but how do we create our own components? The easiest way to generate a new component is to use the CLI and issue the following command: `$ ng generate component name-of-component`. So, to generate a new component named `to-doitem`, we would type `$ ng generate component to-doitem` on our command prompt. Remember to do this from within the `src/app` folder. The CLI will generate this component and insert it into its own folder, and the name of the newly created folder will be the same as the component.

Inside this folder, you will see four new files and their names all start with `todoitem.component` because the name of our component is `todoitem`, and, well, it's a component. We'll discuss what the file ending in `spec.ts` is used for later, but you may already have a good guess as to what the other three files are for. Let's verify what you are probably already thinking; the component file itself is indeed the one named `todoitem.component.ts`. This file contains a reference to two of the others: `todoitem.component.html`, which is the template for the component (the HTML code, for defining its markup structure), and the `todoitem.component.scss` file, which will hold the styling for the component. Additionally, the CLI modified an existing file named `app.module.ts`. We'll discuss this file in more detail later on, but for now, all you need to know is that this file acts as a registry for your application's components.

You may be thinking, *That's a lot of files. Are they all needed?* The short answer to that is no. In Chapter 5, *Flex-Layout – Angular's Responsive Layout Engine*, we'll look at how we can eliminate the `.html` file, and the `.scss` files, and just put all of our component *stuff* (the HTML and the styling) into our component file. However, there is a reason the Angular team provided the mechanism to have all these things be separate—so your application's code can be tidy and organized. You can thank them later.

A nice shortcut syntax when using the CLI to generate a component is to type `$ ng g c name-of-component`, where `g` is short for generating, and `c` is short for the component.

In addition to creating our own components from scratch, which we'll look at in depth in Chapter 5, *Flex-Layout – Angular's Responsive Layout Engine*.

Code listing for the to-do list application

Now that you have the to-do component generated, you have four new files within the `todo` folder. You'll edit three of them to look like the following code listings that follow. You also need to edit three of the files that were already in your project, (where we'll open the curtains to meet the wizard), we can also integrate components from other libraries and frameworks into our application. We'll take a look at how to do this with NG Bootstrap in Chapter 6, *Building Angular Components*, and with Angular Material in Chapter 7, *Templates, Directives, and Pipes*. There's no shortage of components for Angular, and the amount available for your use will only grow over time.

Whenever I learn new technology and follow along with a book, blog post, or whatever else, I enter everything in by hand—even when the files are available for download. Yes, manual entry can be a tedious process, but it engages your brain, and the material and concepts start to get absorbed. Simply downloading the files and cutting and pasting the contents into your application does not have the same effect. I'll let you decide which way you want to go. If you opt for downloading the code, there are instructions at the beginning of this book for doing so:

- The code listing for `todo.component.html` (within the `src | app | todo` folder) is shown here:

```
<div class="container dark">
  <div class="column">
    <p>Add a todo item</p>
  </div>
  <div class="column">
    <p>Todo list ({{ itemCount }} items)</p>
  </div>
  <div class="container light">
    <div class="column">
      <p class="form-caption">Enter an item to add to your todo
list</p>
      <form>
        <input type="text" class="regular" name="item"
placeholder="Todo item ..." 
          [(ngModel)]="todoItemText">
        <input type="submit" class="submit" value="Add todo"
(click)="addTodoItem()">
      </form>
    </div>
    <div class="column">
      <p class="todolist-container" *ngFor="let todoItem of
todoItems">
        {{ todoItem }}
      </p>
    </div>
  </div>
```

- The code listing for `todo.component.ts` (within the `src | app | todo` folder) is as follows:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-todo',
  templateUrl: './todo.component.html',
  styleUrls: ['./todo.component.scss']
```

```
)  
export class TodoComponent implements OnInit {  
  itemCount: number;  
  todoItemText: string;  
  todoItems = [];  
  ngOnInit() {  
    this.itemCount = this.todoItems.length;  
  }  
  addTodoItem() {  
    this.todoItems.push(this.todoItemText);  
    this.todoItemText = '';  
    this.itemCount = this.todoItems.length;  
  }  
}
```

- The code listing for `todo.component.scss` (within the `src | app | todo` folder) is as follows:

```
.container {  
  display: grid;  
  grid-template-columns: 50% auto;  
}  
.column {  
  padding: .4em 1.3em;  
}  
.dark {  
  background: #2F4F4F;  
}  
.light {  
  background: #8FBC8F;  
}  
input.regular {  
  border: 0;  
  padding: 1em;  
  width: 80%;  
  margin-bottom: 2em;  
}  
input.submit {  
  border: 0;  
  display: block;  
  padding: 1em 3em;  
  background: #eee;  
  color: #333;  
  margin-bottom: 1em;  
  cursor: pointer;  
}  
.todolist-container {  
  background: rgb(52, 138, 71);
```

```
padding: .6em;
font-weight: bold;
cursor: pointer;
}
.form-caption {
```

- The following is the code listing for `app.component.html` (within the `src | app` folder). Chapter 1, *Quick Start* : to-do List (quick example app):

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

- The code listing for `app.module.ts` (within the `src | app` folder) is as follows:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { TodoComponent } from './todo/todo.component';
@NgModule({
declarations: [
AppComponent,
TodoComponent
],
imports: [
BrowserModule,
AppRoutingModule,
FormsModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

- The code listing for `styles.scss` (within the `src` folder) is shown here:

```
/* You can add global styles to this file, and also import other
style files */
body {
font-family: Arial, Helvetica, sans-serif;
color: #eee;
background: #869bbd;
padding: 4em;
}
a {
```

```
color: #fff;
text-decoration: none;
}
ul {
list-style-type: none;
margin: 0 0 2em 0;
padding: 0;
}
ul li {
display: inline;
margin-right: 25px;
}
ul li a {
font-size: 1.5em;
}
```

Cool! So, now you have all the code in place. Do you remember how to run your Angular application? Enter `$ ng serve` at your command prompt, and once the message comes up that the compilation was successful, open your browser and go to `http://localhost:4200`. Does the application work? If so, congratulations on building your first Angular application! If not, check for typos.

Play around with your new application. We haven't bothered to take the time to add functionality to edit the to-do item, or to delete them, but you can clear it out by just reloading the application by hitting your browser's refresh button.

Why do things get cleared out upon refreshing the page? This happens because we have an SPA and are not persisting the data that we enter into a database. We'll definitely be sure to add the ability to persist our data when we build our much larger application, which will be introduced to you by the end of this chapter.

Interpolation

Interpolation is how you get a value from a variable within your component class to render in the component's template. If you recall, the logic for the component goes in the export section of the component class. That is the same place where variables are that you would like to use interpolation to have their values rendered in the template (that is, rendered on the web page). Let's assume that you have a variable called `items` and that its value is currently 4. To render the value in the template, you use a pair of double curly braces with the variable in between them. The variables and component logic are written inside the class.

Don't worry—we'll see lots of code snippets throughout the book that use interpolation, but for now, you can see this sample code that shows it in action. The code is meaningless and hardcoded for now, but it does demonstrate interpolation.

The first screenshot is of the component file (`home.component.ts`); the variable is declared on line 10:

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

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  items: number = 4;

  constructor() { }

  ngOnInit() {
  }
}
```

This second screenshot is of the component's template file (`home.component.html`). Notice the pair of double curly braces on line 6:

```
<div class="container dark">
  <div class="column">
    <p>Add a todo item</p>
  </div>
  <div class="column">
    <p>Todo list {{ items }} items</p>
  </div>
</div>
```

This last screenshot shows the rendered value, 4 in this case. That's the basics of interpolation. We'll see much more advanced usage of it throughout the book as we work on our annotated photo album:



Templating and styling

We've already mentioned something about templating and styling in the last few paragraphs of the *Components* section. Now that we have a small project available to us—the one that we created with the CLI—we can see what this looks like in code. Open your Angular project in the IDE, and open the `app.component.ts` file. Lines 5 and 6 in this app component file contain the references to its associated template (the `.html` file), and its style file (`.scss`), respectively. The following is a screenshot of the open project in my IDE, with the `app.component.ts` file open:

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files and folders: `app.component.ts`, `app.module.ts`, `MINIAPP` (which contains `home`), `app-routing.module.ts`, `app.component.html`, `app.component.scss`, `app.component.spec.ts`, `app.component.ts` (which is the active file), and `app.module.ts`. The main editor area displays the `app.component.ts` file content:

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

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

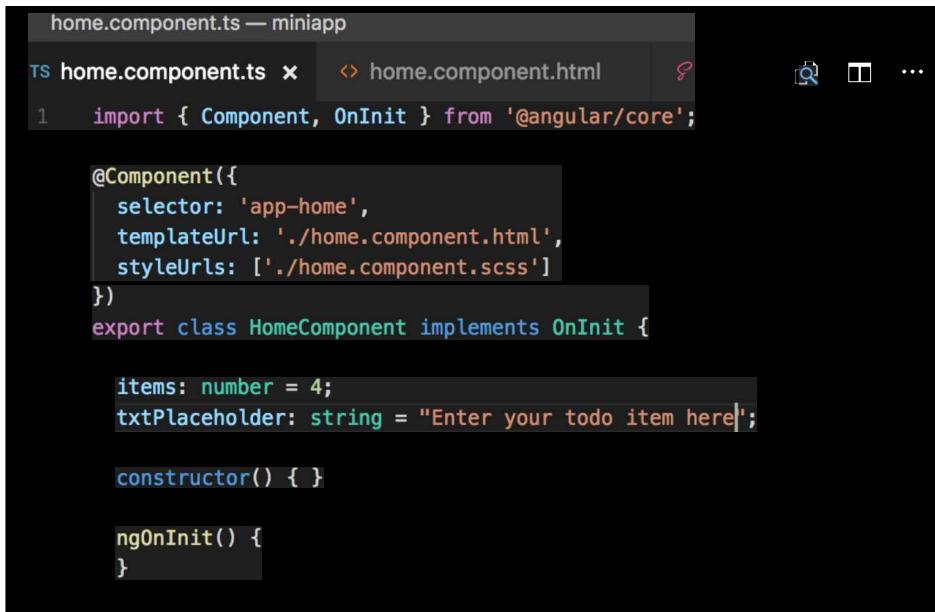
Property binding

There are two types of data binding we can do in Angular, namely, one-way and two-way. Interpolation is very similar to one-way data binding. This is because, in both cases, data flows from the component class to the component template and not the other way around. Property binding is data binding in the sense that the data is being bound to a property.

It's also possible to have two-way property binding—meaning, not only can the value of a component property be bound to the template, but the template can alter the value of a component property. This is made possible in Angular by `ngModel`. Again, don't worry about this for now. We will take a look at two-way property binding later on. Just know that both one-way and two-way property binding is possible in Angular.

Implementing one-way property binding is very straightforward. All you need to do is to put square brackets around the HTML property (in the component's template) you want the data bound to, and assign the variable to it. To see a quick example of what one-way property binding looks like in code, take a look at the next three screenshots.

The first screenshot is of the component file (`home.component.ts`); the variable, `txtPlaceholder`, is declared on line 11:



```
home.component.ts — miniapp
TS home.component.ts ✘  ↗ home.component.html  ⚡  ⌂  ...
1 import { Component, OnInit } from '@angular/core';

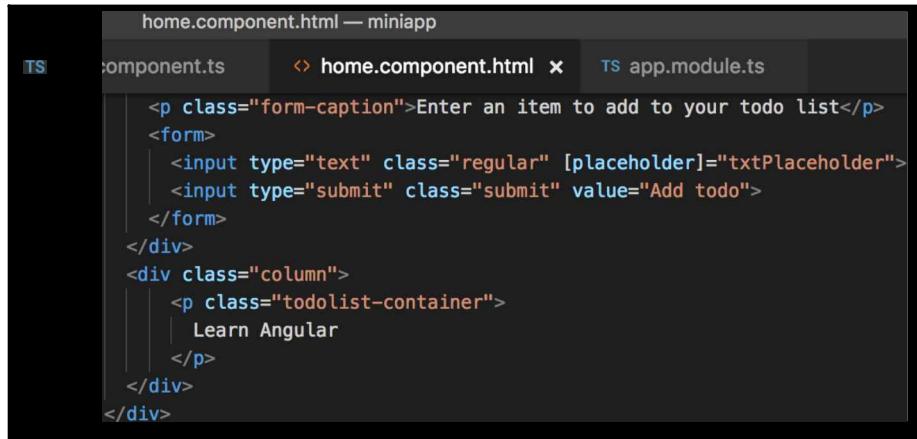
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  items: number = 4;
  txtPlaceholder: string = "Enter your todo item here";

  constructor() { }

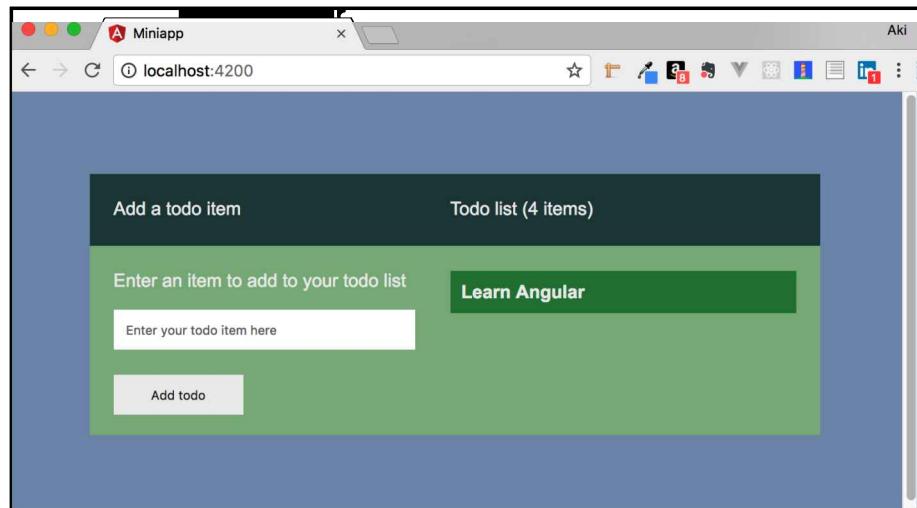
  ngOnInit() {
}
```

This next screenshot is of the component's template file (`home.component.html`). In line 14, you can see the square brackets around the placeholder property of the input element:



```
home.component.html — miniapp
TS component.ts < home.component.html x TS app.module.ts
<p class="form-caption">Enter an item to add to your todo list</p>
<form>
  <input type="text" class="regular" [placeholder]="txtPlaceholder">
  <input type="submit" class="submit" value="Add todo">
</form>
</div>
<div class="column">
  <p class="todolist-container">
    Learn Angular
  </p>
</div>
</div>
```

This last screenshot is of the application running in the browser. You can see the text, **Enter your todo item here**, inserted as the placeholder text for the textbox via one-way property binding:



Event binding

In Angular, event binding simply means that an event will be registered on an element within a component, and when that event happens, it will trigger a function to be called. In other words, an event will call a function. There are a ton of events that you can have Angular listen for—such as a button being clicked, a mouse hovering over an image, or when the user presses a key down when the cursor is in a textbox, and lots more. And, of course, you can write any function you can possibly think of to make other stuff happen, such as calling a web service, changing the color of the background page, calculating the value of Pi to 1,000 decimal places, or virtually anything else your heart desires. But how can we set up event binding in our Angular application to tie the event we're interested in, on the element we're interested in, to run the function we want? Well, thankfully, the Angular team made this super easy for us.

Suppose we'd like to have some user interaction via events such as click or mouseover—we can use event binding to map the functionality.

Now, that example is not very interesting, but we do have our to-do list application to look at the code we've already written. If you've typed in the code listings.

Our sample project

Learning a new programming language, or learning a new framework, is a matter of hands-on experimentation and repetition. Even Dr. Sheldon Cooper from *The Big Bang Theory* can't learn Angular just by reading a book on it. However, just following along with random code examples isn't much fun to do and, at the end of the day, you don't really have anything you can use. So, the approach we'll take on our journey to Angular mastery is to build a fully fledged web application that is fun to build and practical since you can deploy it and use it yourself.

Annotated photo album

The application that we'll be building together is based on one of the online services I've launched called Vizcaro. Vizcaro is a photo sharing service, but instead of sharing individual photos, you share albums (groups of photos). Also, the photos and albums will be annotated so you can add titles and captions to them. Our version won't have all the bells and whistles that my online service offers, but it will have just enough parts to make it a great web application to build in order to learn the material in this book.

Design principles

There are generally two types of design: the way in which you design your user interface (the GUI), and the way in which you design the software components (API interfaces, services, components, and more). Throughout this book, we'll be covering quite a few design principles for the code. Angular is a spectacularly well-designed piece of software, and this is great for us because it provides a perfect opportunity to discuss software design as we learn Angular itself, in addition to when building our application. We'll also be covering user interface design principles in general as we build our application throughout the remainder of this book, but particularly when we build out our templates using our wireframes to help guide us.

In general, the term UX design is used when discussing user interface design. Borrowing a definition of UX design from *Wikipedia*:

"UX design is the process of enhancing user satisfaction with a product by improving the usability, accessibility, and pleasure provided in the interaction with the product."

This is a good definition and applies to more than just software products.

Wireframes

Wireframes have been around since the early '80s. Their focus, at least initially, was on what the screen in the desktop application did (remember, web applications were not around yet), and for its general layout. They were not meant to be used as what the final design was to look like—including font selection, colors, and other properties of the controls on the screen. In essence, they were the *prototype on paper*. *Paper prototyping*, conversely, is a process that uses the wireframes. It's worth noting that the nouns *wireframes* and *mockups* are used interchangeably—they are the same thing. I'll briefly cover the paper prototyping process at the end of this chapter.

Wireframing tools

As you probably have already guessed, or already know, there are several tools available for creating wireframes when laying out your application, such as Balsamiq Mockups, Mockflow, and Visio. For my web applications, and in this book, my preference is to use Balsamiq Mockups. Which one you end up using for your applications, or are already using, doesn't matter. In fact, even if your wireframes are drawn by hand using a pen on the back of your napkin from your favorite fast food restaurant, it would be cool with me. Seriously, the important thing is that you get into the habit of creating wireframes before writing a single line of code. Why? Because it's a smart thing to do, and it saves you a lot of time. Additionally, it gives you the perfect opportunity to really think about what you're going to be building. And, it's something that you can present to users in order to get their feedback on the usability without writing a single line of code. There are even more benefits; it gives you some idea on how you would design your data model for the application, as well as APIs for services it may consume. Would you start a business without a business plan? Would you build your dream home without a blueprint? Building a web application shouldn't be any different specification out the pages using wireframes. Always. Capiche?

Wireframes for our annotated photo album

There are 10 wireframes that we'll be using for building our application—one for each screen it will have. The following is the listing of them, and a short description precedes each screenshot.

Home page

Every web application needs a starting page of some kind. It's known by many names, typically one of these: home page, landing page, index page, or splash page. Ours will be straightforward. No Flash animation or rainbow-colored backgrounds; it will be a simple page that lets the user know what the site does, and hopefully, it does that within five to seven seconds. If it doesn't, you may lose the visitor forever:



Dashboard

Most web applications don't have a dashboard page, but those that do typically provide a summary of *things* the user has, the last time they had logged in, and any notifications that the company would like to bring to the user's attention. If you use online banking, chances are that your bank's online banking web application has a dashboard page—and it probably is a list of accounts (checking, savings, credit cards, car loans, and more), and the balances on those accounts that you have with them.

We're going to build an application that users will implement to create photo albums, and so our dashboard will contain the number of photos we have uploaded, the number of photo albums, the last time we logged in, and more:



Image upload

Since our application is supposed to enable our users to create photo albums, we'd better have a way for them to upload their photos! Yes—we are dedicating an entire web page to upload one photo at a time because we will use this same page to preview it after it has been uploaded—and to *undo* the upload. You'd be surprised to know that there is a well-known photo sharing site that does not show what you have just uploaded until you go to the listing of your photos! Having an immediate confirmation that the photo you intended to upload is, in fact, the one that was uploaded:

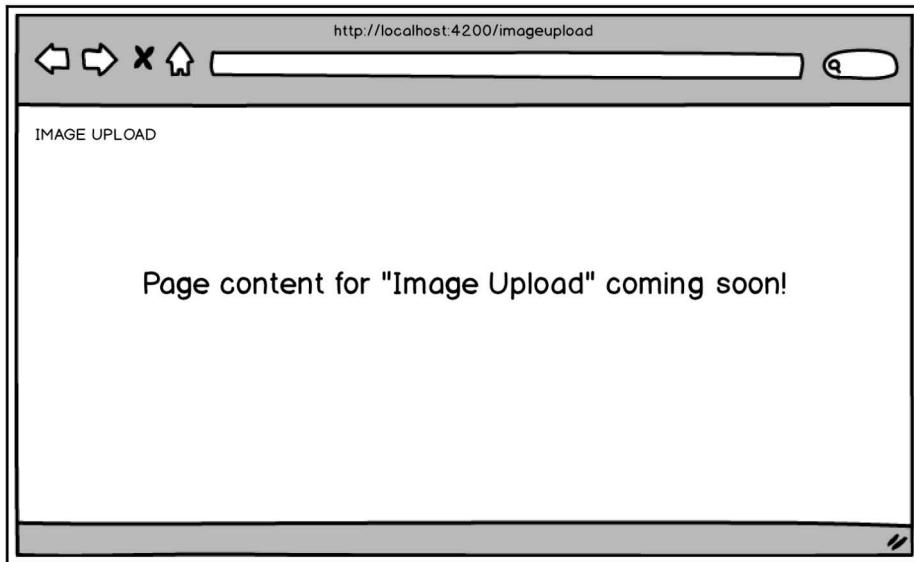


Photo preparation

Uploading a photo is the first step in our annotated photo album application. We are devoting another web page to *preparation* the photo. This is where we will allow the user to resize the image and annotate it (give it a name and a caption). The photo's caption will be shown when viewing it in the photo album:



Create Album

Before the user can add photos in their photo albums, they have to be able to create the albums. This is what the following web page will be used for:

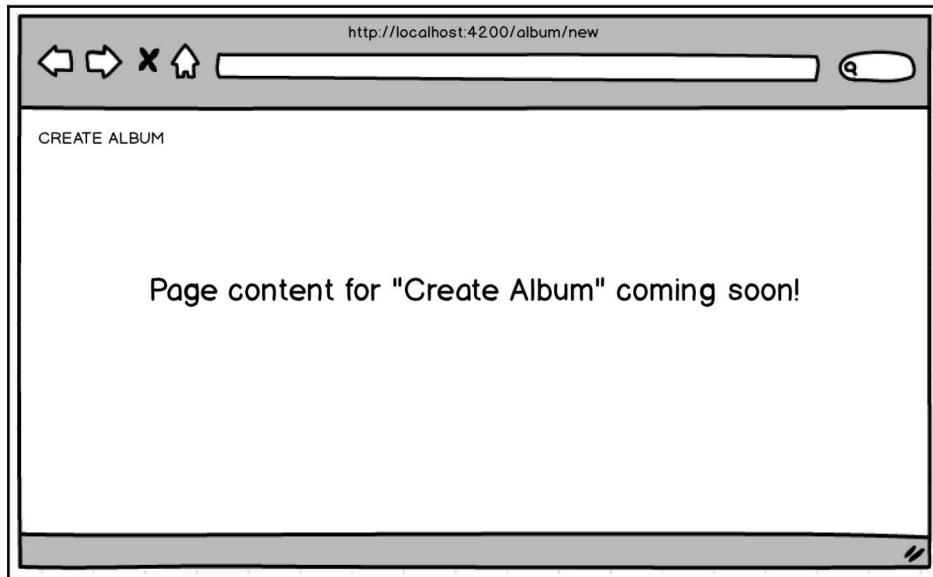


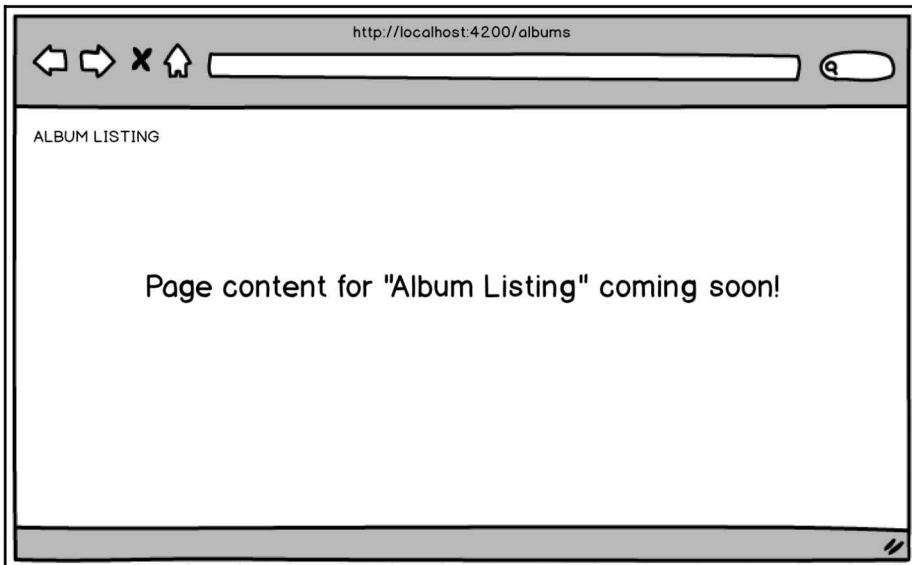
Photo listing

You always need to think of usability and how to design the most intuitive user interface you possibly can. This page will show a listing of all the photos that the user has uploaded. Additionally, they can edit the name and caption for any of the photos right on this same page. The less jumping around from page to page that your users need to do, the happier they'll be:



Photo album listing

This page does for photo albums what the previous page did for photos—provides a listing of all the albums the user created and has an intuitive way to edit their name and description (as opposed to the caption for photos) without going to another web page:



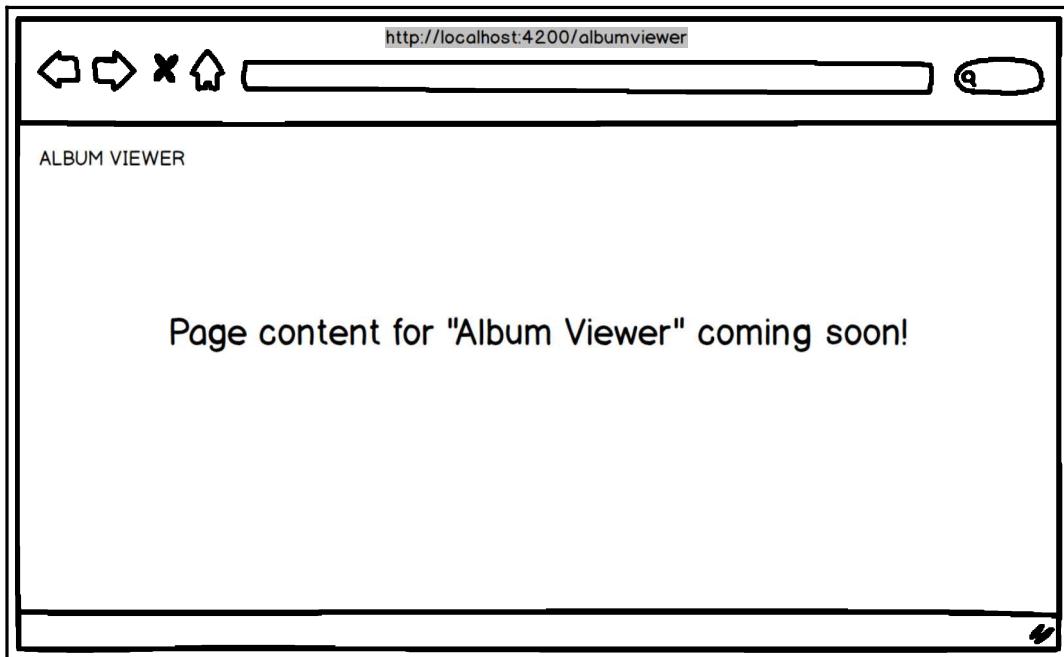
Workbench

The workbench is the place where the user will be able to drag a photo onto a photo album. This will be our way of allowing the user to intuitively associate a specific photo to a specific album. By the way, not only will our drag and drop feature be intuitive and functional, it will add an element of fun for the user. Psychologically, users want to *play* on the site. Dragging and dropping—though not a mind-blowing experience—is more fun to do than it would be to select a photo from a photo dropdown, then select an album from an album dropdown, and, finally, click a *connect* or *associate* button. The former method would please the user, and the latter would make them send you a nastygram—then leave the site, never to return:



Album viewer

At the end of the day, the users want to see their photo albums in an engaging way. The drag and drop stuff was fun but they're not here for that. They're here to see photos from their son's birthday party, their daughter's high school graduation, or pictures of their dream home. This is an important page for them; it's where their work of using our site will pay off for them. Let's not disappoint them:



This concludes the introduction of our annotated photo album that we'll be building throughout the remainder of the book, the wireframing and this chapter's planned material that was to be covered. I did, however, want to quickly discuss paper prototyping, as a closing to the chapter, and to tie it back to the planning of our Angular application.

Paper prototyping

As mentioned near the beginning of this chapter, paper prototyping is a process. We had also mentioned that the core focus of paper prototyping was on usability testing. What we hadn't mentioned was that paper prototyping should be a part of the software development methodology that your development team uses—be it waterfall, or some form of agile. Specifically, paper prototyping should come immediately after the requirements documents are delivered to the project manager. That is the high-level view of what paper prototyping is and where it fits in.

Let's now take a look at the mechanics of the process at the lower level, meaning the interaction of the development team with the users of the application that is to be developed.

The mechanics, or process in which paper prototyping is done is to first create the wireframes and print them out (I know, more trees will be cut down and global warming becomes an even larger threat, but paper prototyping is important). Once the paper version is in front of you, your boss, your client, or a group of intended users (such as a focus group), you, or whomever else, will *use* the paper prototype just like they would by clicking their mouse onto the actual web application as if it were already finished and in production. You would ask them to pretend that it was the actual finished application. It sounds silly at first but human beings have an incredible imagination, and with little effort, they will be using it as they would the real thing! This is not hypnosis here but rather something quite magical starts to happen. Without your direction, except for at the very beginning when you explain what you're requesting of them and why, they will start to ramble on, talking out loud about the actions they are taking, or are thinking of taking, such as, *Ok, so now I need to fill this out and submit the form, or Where is the button to undo what I just did. I made a mistake.* The best information you're going to get from people going through this exercise is when they offer suggestions for how something may be improved, such as *It'd be nice if I can easily navigate back to where I... .* Can you imagine coding web pages and then realizing the changes being asked to have a far-reaching effect and would be a time-consuming change? It happens a lot.

Do you realize what you have by doing this? You have test users and you haven't written a single line of code! It is quite a powerful exercise! Try it, and email me your story.

So, when I explain this to a client—not to be the user, but how to present the paper prototyping process to their users and/or clients, I'm usually asked, *But this is paper. How do we change the screen?* I reply the best way I can—by showing them an example. I usually have a sample set of wireframes with me. Not only to demonstrate the paper prototyping process but also to just show a good example of wireframes. I lay down the login screen on the table we're seated at, and I ask them to log in using their finger as the mouse pointer, and then typing on their imaginary keyboard. After they chuckle and just play along with me by typing their username and password by fake typing on the table under the wireframe, they then hit the login button, and I become the computer—I pick up the login wireframe and put down the dashboard wireframe. They usually stop chuckling, look at the dashboard page, take a few seconds, and then look at me nodding and say, *This is cool. I get it.*

Summary

This chapter was packed with a wide array of topics, I know. It was inevitable because there is no one best starting point for all the readers. For instance, some readers know what wireframes are and have used them for years, while other readers may have only just heard of the term, or maybe not even that. This is the third edition of this book, but it is quite a bit different from the first two editions and even if it was largely the same, which it isn't, it doesn't mean that readers have already gone through the first two editions. You can think of this first chapter as a type of funnel—a wide enough funnel that brings readers of all levels of experience, and differing knowledge, into a common track for learning Angular, and the other symbiotic technologies that are covered in this book. Starting with Chapter 2, *ECMAScript and TypeScript Crash Course*, the funnel approach is over. The rest of the chapters will be a lot more focused on the subject matter at hand. So, thank you for hanging in there with me. Still, I hope that there were a few things that made wading through this first couple of dozen pages worthwhile, even if you're not completely new to Angular.

In review, we covered the evolution of Angular, including its semantic versioning and release schedule. Although the installation of NodeJS, npm, and the CLI are covered in Appendix A, this chapter is what guided that discussion, and we then used the CLI to build our first Angular app and a to-do list app together. We'll name the app to-do list because we're developers and not marketers (wink). We also covered how to use StackBlitz for building the same Angular application without having any reliance on our local development environment. We then covered the first very basic building blocks of Angular that you need to know well since they will be used again and again for any Angular apps you build. Namely, these were templating, property binding, event binding, and class binding. Lastly, we introduced the annotated photo album application that we'll be building together throughout this book and covered UX design principles, wireframing, and paper prototyping along the way. Whew! Mama mia!

In the next chapter, we will first understand the relationship between JavaScript and TypeScript. We will then, as the name suggests, do a crash course on TypeScript and its advantages over JavaScript.

2

ECMAScript and TypeScript Crash Course

Chapter 1, *Quick Start*, was a heterogeneous mix of topics that may have seemed a little loosey-goosey, but it was presented that way to lay down a swath of material and topics related to frontend web application development—with an obvious inroad to Angular, for starting your adventure in becoming an Angular guru. From this point onward, each chapter will remain as focused as possible and thus is dedicated to a specific area of coverage. As you progress through the chapters, you will generally find them to become more and more technical. This is a natural progression and not to be feared, because, as you may recall, one of my promises made to you in Chapter 1, *Quick Start*, was to not get so deeply entwined in the technical details as to reach a point of diminishing returns. In other words, no deep technical babble that won't add any value to our purpose. Instead, we'll get as technical as we need to get—no more, and no less. Additionally, the material will be presented in an engaging way, where you have the greatest chance for retention with the least amount of effort.

Regardless, please do not take this comment to mean that you don't have to work at it. As with anything else in life, the better you want to become at something, the more work it will take on your part. We all reap what we sow. That being said, this chapter and the next one will be a gradual ramp up to the technical deep dives that follow—kind of like the warm-up before we start working our Angular technical muscles throughout the rest of the book.

This chapter will cover the following topics:

- The relationship between JavaScript and TypeScript.
- A crash course on TypeScript

The (quick) roadmap

This chapter is a crash course on TypeScript and is meant to serve as a way to fast-track the transition from JavaScript to TypeScript for developers who are already comfortable with JavaScript.

As was mentioned in [Chapter 1, Quick Start](#), TypeScript is the language we'll be using throughout this book when working on Angular-specific things and so this chapter serves as your preparation for the programmatic part of getting up to speed for web development with Angular. You can think of [Chapter 3, Bootstrap – Grid Layout and Components](#), as being this chapter's cousin, in that its goal is similar but for the presentation side (that is, the layout of the web pages) as opposed to the programmatic side. Together, [Chapter 2, ECMAScript and TypeScript Crash Course](#), and [Chapter 3, Bootstrap – Grid Layout and Components](#), will round out the prerequisites for building client-side web applications in general—regardless of the client-side web application framework, but also specifically for Angular-based ones. From [Chapter 5, Flex-layout – Angular's Powerful Responsive Layout Engine](#), onward, it's going to all pretty much be Angular-centric. In a nutshell, that's our roadmap for this chapter and the next. Let's start warming up our technical muscles!

The relationship between JavaScript and TypeScript

JavaScript and TypeScript are inextricably tied together. Thus, while this chapter covers two technologies, ECMAScript and TypeScript, they are similar enough for this chapter to cover both simultaneously. How similar are they to each other? Well, for the most part, you can consider TypeScript to be a superset of JavaScript. The most useful description of their relationship is the following: TypeScript is a strictly typed language with lots of powerful features with optional typing, and with its transpiler, it becomes plain JavaScript. This is important and brings several advantages for developers; it's compelling enough for Google's Angular team to switch from JavaScript to TypeScript for developing Angular itself. We'll cover what a transpiler is, as well as what the advantages of using TypeScript are, very shortly.

A series of fortunate events for JavaScript

Before we jump into the technical parts and the code, it would be worthwhile to take a quick look at the evolution of JavaScript, and some of the drivers that led to the need for a language such as a TypeScript. Also, just as Angular's naming jungle caused some confusion in the development community, JavaScript has had an even more confusing versioning past since its inception over two decades ago, and so I'd like to try and clear up some of the confusion around JavaScript's version naming. More importantly, I'd like to cover what I like to refer to as the series of fortunate events for JavaScript. This will help set the pace for much of the material we'll be covering together in the rest of the book.

I must confess, I love working with JavaScript. I have always enjoyed the language—not for the language itself, but because it allowed us to make the web come alive without the need for other plugins, such as Flash or Shockwave. However, in recent years, there are a few additional reasons for why I love working with the language, and the exact reasons I love JavaScript are precisely the series of fortunate events that I will cover shortly. Having said that, I have friends in the industry that are on the other end of the spectrum, who view JavaScript as a *toy language*, and prefer to remain shackled to languages such as Java and C#, avoiding JavaScript at all costs until they have to grudgingly write some code for the client side. These old-timers' usage of JavaScript typically extends no further than binding a click event to a function call (using the jQuery library) to submit form data to their Java or C# APIs. Sure, about a decade ago, JavaScript wasn't as powerful as Java or C# for a few reasons, such as it was a language that only ran on the client side (that is, on browsers), there were not as many libraries for it, and highly performant runtimes for it did not really exist. All this was about to change due the series of fortunate events—specifically, three of them. Let's review them quickly.

Chromium project

The first one was Google's Chromium Project. In September of 2008, Google released Chrome V8, which was a high-performance JavaScript engine. Google's Chrome V8 supercharged the way in which JavaScript code was able to be executed. It was such a successful project that it enabled other technologies to materialize, including the one that truly changed JavaScript's future immediately and forever: Node.js (referred to simply as Node).

Javascript frameworks

The second event in the series of fortunate events that have cemented JavaScript's reign as the most important programming language for web applications, and possibly for mobile and even desktop applications as well, has been the explosion of JavaScript frameworks. Ever since about 2010, the development world has gone absolutely ballistic in its insatiable urge to create JavaScript-based frameworks, not only for client-side web application development (such as Ember, Knockout, and React), but for server-side libraries (thanks again to Node), frameworks for creating native mobile applications (such as Ionic, React Native, and Native Script), and frameworks for developing desktop applications as well (such as Meteor and Electron). I haven't mentioned Angular in this paragraph, because we've already learned that Angular can be used for building cross-platform applications for all three domains—browser, desktop, and native mobile.

ECMAScript 2015

The third event in this series of fortunate events was the release of ECMAScript 2015. ECMAScript is the official name for JavaScript when discussing its standard. Despite the increase in major version releases, the JavaScript language had largely gone on unchanged for many years. This was due to differences between influencing players (who are best left), causing splintered development and stalled progress in evolving the language.

To summarize, this was a whirlwind tour of the current state of JavaScript and some of its ecosystem. JavaScript's ecosystem is so vast that several books would need to be written to cover it. For instance, we've not even made mention of visualization JavaScript libraries. There are hundreds, if not thousands, of libraries available for JavaScript that you can use for your projects that we can't even hope to begin to cover. However, there is a part of the JavaScript ecosystem that we will absolutely cover: unit testing. You probably know about the importance of unit testing, and have likely written unit tests for server-side code using frameworks such as JUnit, NUnit, RSpec, and others, depending on the programming language you used. However, unit testing is just as important on the client side, and most developers don't do it, even though they may write unit test script for the server side. Well, in Chapter 13, *Unit Testing*, you will learn how to write unit tests for the client side and, specifically, how to write them to test your Angular application. The two frameworks we'll cover together are Jasmine (a popular unit testing framework) and Karma (a test runner that has plugins for testing frameworks, such as Jasmine).

With this chapter's technical preamble out of the way, let's strap on our scuba gear and dive into TypeScript's ocean!

TypeScript crash course

TypeScript has a number of advantages over JavaScript for developers, including the following:

- Purely object-oriented
- Optional static typing
- Type inference
- Access to ECMAScript features
- Transpilation
- Great tooling support with IntelliSense
- You can build Angular applications!

Transpilation versus compilation

Developers can usually define what compilation is, within the context of programming. The definition would be something like this: compilation is the process of transforming source code, by putting it through another program known as a compiler, into machine-readable code. This resultant code is typically referred to as assembly code, which is a set of machine instructions native to the machine's CPU, which the code is meant to be run on.

Transpilation, on the other hand, is the process of transforming source code written in one language into equivalent code in another (or target) language. While this definition is good enough for discussion, for it to be fully accurate we have to also note that the source and target languages may, in fact, be different versions (or releases) of the same language. For our transpilation needs, we'll be using TypeScript's transpiler, tsc, which comes packaged with TypeScript. The reason we care about transpilation when building Angular applications is because we will be writing our code in TypeScript. However, web browsers only have JavaScript engines/interpreters, and so we need a way to transpile it into JavaScript.

let

The `let` and `const` keywords were introduced in ES6. In order to discuss what these are and how they work, let's review what the `var` keyword does. Prior to ES6, the way in which you initialized a variable was to use the `var` keyword. The two things to remember about `var` are the following:

- When you use `var` to define a variable outside of a function body, it becomes globally scoped. This is to say that all the other functions in your JavaScript file have access to it. While this may sometimes be convenient, it can also be dangerous because the value may inadvertently be changed by a function other than the one that you intended the variable to be used for. This is possible when more than one function refers to the same variable name.
- When you use `var` to define a variable within a function, it becomes locally scoped. In contrast to globally scoped variables, locally scoped variables are only accessible within the function in which they were created. This is true regardless of block scope because JavaScript variables declared with the `var` keyword are scoped to the nearest parent function.

You can, of course, still use `var` to declare and define your variables, since the keyword has not been deprecated. It's just that you now have more explicit control over the behavior of your initialization code, and the readability of the code has improved with `let` and `const`, since the intentions are clear for when you're looking at JavaScript code.

The `let` keyword creates block scope local variables and gets its name from other languages that have a similar construct, such as Lisp, Clojure, Scala, and F#. In these languages, variables declared using `let` can be assigned, but not changed. However, in ES6, variables assigned using `let` can be changed; even so, regardless of whether they are changed or not, the variable is a local block scoped variable.

If you find this is a little confusing, you're not alone. Getting a firm understanding of the nuances in variable scoping is not something that you can learn by just reading.

Programming is like learning math (or like learning most things, for that matter): the more you do it, the better you get. That being said, one way to boil it all down in your mind is to look at this one main difference between `var` and `let`: since you can have more than one block within a function, and even nested blocks (or sub-blocks), variables defined with the `let` keyword are accessible only within the block they are defined, as well as from that block's nested blocks. In contrast, the scope of variables defined with `var` is the entire closing function. Remember that one main difference, and you're golden.

Let's look at some code to understand the impact of the `let` keyword, and then we can move on to discussing the `const` keyword:

```
let x = 5;
if (x === 5) {
  let x = 10;
  console.log(x); // this line will output 10 to your console
  // note, x was allowed to be changed
}
console.log(x);
// this line will output 5 to your console
// because the change to x was made within a block
```

Const

The `const` keyword creates a constant. You'll be glad to know that since you've gone through the pain of understanding what the `let` keyword does, understanding what the `const` keyword does will be dead simple. Ready? Here it is ... `const` and `let` are identical in the way in which their scoping works. The only difference between `let` and `const` is that you cannot redeclare a constant, and its value can't be changed. That's it. Let's move on to discussing some data types.

Data typing

Every programming language has data types. They only vary in the number of types available, and the values (and range of values, for number types) that the typed variable can hold. I won't wade into the philosophical debate between strongly typed versus statically typed versus loosely typed languages (usually referred to as static versus dynamic typing) in this book—but since this chapter is devoted to JavaScript and TypeScript, I do need to say a couple of quick things about their typing.

JavaScript is a loosely typed language—which is to say it is a dynamic language as opposed to it being a static language. What this means is that variables in JavaScript are not bound to any particular type, but rather their values are associated with a type. Variables can be assigned, and re-assigned, to values of all available types. While convenient, hard-to-find bugs can occur, since there is no compiler that checks for adherence of value to a typed reference—and this is because when you declare a variable using `var`, `let`, or `const`, you do not specify an associated type.

In contrast, TypeScript is optionally statically typed. The important word here is *optionally*. TypeScript is a statically typed language, yes, but it does not force you to explicitly annotate your variable with the intended type. This is because TypeScript has what is called type inference, which is to say that the TypeScript runtime will infer the variables data type at runtime. This is the default behavior of TypeScript. Now, this is where the optional part of it comes... if you want to strictly type the variable, thereby binding the datatype to the variable instead of it resting with the variable's value, you have to explicitly add a type annotation to the variable declaration.

Here it is in code:

```
var num: number = 12345; // this is a statically typed variable  
var num = 12345; // this is a dynamically typed variable
```

Both the preceding lines are valid TypeScript syntax, but here are their differences:

- The first line, the statically typed variable, which is annotated with the `num` keyword, is checked by the TypeScript transpiler, and any issues will be reported by it
- The second line, where the variable declaration is made in the JavaScript fashion (that is, no annotations for static typing), goes unchecked, and any issues will only be found at runtime

ES6 has seven data types, six of which are known as primitive data types, and one of which is a reference data type (which is just called `Object`). JavaScript also has several built-in data types in its standard library, but since this is not comprehensive coverage of JavaScript, we'll only cover a few of them here: the ones that you're likely to use in your Angular development.

The following is the list of the primitive data types provided:

- `Null`
- `Undefined`
- `Boolean`
- `Number`
- `String`
- `Symbol`

The following are the built-in data types provided:

- Dates
- Arrays
- Maps
- Sets

Objects

Only having primitive data types and built-in complex data types is not enough when writing expressive software that attempts to model the real world or fictitious worlds (in gaming). The solution is to have programming languages with a construct for creating custom objects. Fortunately, JavaScript, and thus TypeScript, is a programming language that allows for the creation of custom objects. An object in JavaScript is a collection of mapped keys and values, where a key can be either a string or a symbol. This is similar to the case for many other programming languages, such as for Python's dictionary, and Ruby's hash. Not to get too technical just for the sake of being technical (which is a pet peeve of mine, and maybe is one for you as well), but JavaScript is not a *classical* object-oriented language. Rather, JavaScript uses prototypal inheritance to create other objects instead of creating an instance of an object from a class definition. In other words, JavaScript doesn't have the notion of classes. JavaScript has prototypes. Objects in JavaScript inherit directly from other objects. In fact, when you create an empty object in JavaScript using curly braces, it's really syntactic sugar for using the `Object.create()` method of the built-in `Object` object. There are several ways available to you for creating an empty object in JavaScript. We won't cover them all here, but we'll cover two of the ways that have been available in JavaScript for many years, and are made available to us in ES6:

- Using the `Object` constructor: `var myObject = new Object();`
- Using the curly brace syntax: `var myObject = {};`
- Using ES6 class syntax (we'll get to the syntax in the following *Classes* section)

The first two methods create an empty object. If you want to create an empty object in JavaScript with minimal fuss, the second approach is obviously the easiest. However, the third approach, ES6 class syntax, is what we'll be using in this book.

JSON

JSON, which is an acronym for JavaScript Object Notation, is not a data type per se, but rather structured data. JSON is used as a lightweight data interchange format and is used by many programming languages. Not only will we cover this more a little later, but we will make extensive use of this format to pass data back and forth between our Angular application and the backend web services that we'll also be building for it. Just as programming languages have data types, data interchange formats often do as well. Here are the data types allowed to be represented in JSON:

- String
- Number
- Object
- Array
- Boolean
- Null

You may have noticed that there is a large overlap between JavaScript and JSON data types. This is not by accident because JSON is JavaScript Object Notation, and thus was modeled after JavaScript's data types. Here's an example of JSON data that contains the names and ages of three people (each of which is a JavaScript object):

```
{  
  "people": [  
    {"name":"Peter", "age":40},  
    {"name":"Paul", "age":42},  
    {"name":"Mary", "age":38}  
  ]  
}
```

In the previous JSON example, I have `people` as the key, and its value is an array of three `people` objects. There's no hard and fast rule that says you have to name structures nested structures, but it does become more legible. In this simple example, you could have instead omitted the key with no loss of data, as this next JSON sample shows:

```
[  
  {"name":"Peter", "age":40},  
  {"name":"Paul", "age":42},  
  {"name":"Mary", "age":38}  
]
```

However, the first example, where we have the `people` key, is not only easier to read but is also easier to work with in the code. When we write our RESTful web service APIs for our application in Chapter 12, *Integrating Backend Data Services*, we will take the first approach, providing keys for our collections of data.



Here's an interesting note for you about data interchange formats. While there are a few formats to choose from, such as XML and SOAP (Simple Object Access Protocol), when developing web services, JSON is the most popular one of all, and it was inspired by JavaScript.

Where would we be without JavaScript? I shudder to think about it.

JavaScript runtime environment

The rest of this chapter has many code snippets, so if you'd like to experiment with the material as you work your way through the chapter, it's a good idea to fire up your JavaScript runtime environment. Unless you're using a JavaScript IDE, such as WebStorm by JetBrains, you have a few options available to you for testing out JavaScript code. Here are three of the many choices:

- You can use an online a JavaScript console, such as <https://es6console.com/>.
- You can use Node right in your Terminal (Appendix A shows you how to install Node).
- You can use the console within your developer tools in your browser. For instance, I primarily develop using Chrome, and Google has excellent tools for developers.

Any of these choices will work just fine. I prefer using the Node Terminal for quick little things and it's what I used to test the code I wrote for this chapter.

Arrays

Arrays are part of a collection of objects, and are referred to as indexed collections. If you've written any amount of JavaScript, you have used arrays. Array objects can hold any valid JavaScript data types and can grow and shrink by calling their built-in methods, such as `push` and `splice`. You can search for the existence of a value in the array with the `indexOf` method, get an array's length by using its `length` property, and so on. The JavaScript syntax for creating an empty array is the following:

```
var myDreamCars = [];
```

You can then use the array's built-in `push` method to add an item to the array, like this:

```
myDreamCars.push("Porsche");
```

Alternatively, you can create the array in place in one fell swoop, like this:

```
var myDreamCars = ["Porsche", "Mercedes", "Ferrari", "Lamborghini"];
```

The `indexOf` method is quite handy, and we're sure to make use of it later. Let's quickly cover this one before moving on to `TypedArrays`. When you need to find where in the array a specific item is, or whether it exists in an array at all, you can use the `indexOf` method. Let's assume we'd like to see where in array the Mercedes is. We can search for it like this:

```
var indexOfMercedes = myDreamCars.indexOf("Mercedes");
```

Given our `myDreamCars` array, the `indexOf` function would return 1. This is because arrays in JavaScript start their index at 0, and the Mercedes was in the second slot of our array. What if what we're looking for is not in the array? Let's see what happens when we look for a Corvette:

```
var indexOfMercedes = myDreamCars.indexOf("Corvette");
```

When the preceding line is executed, the `indexOf` function returns a -1, which indicates that the item we were searching for was not found.

TypedArray

`TypedArray` is used in ES6 and even though it has a few of the same methods as normal JavaScript array objects, it is quite different from what you'd probably expect. In fact, `TypedArray` is not an array at all. If you try passing in `TypedArray` to `Array.isArray()`, you will see that the value returned is `false`. OK, so what are they then? `TypedArray` gives us a view of an underlying binary data buffer and enables us to access and manipulate data. We won't be covering `TypedArray` in this book because we won't be using them, and it is an advanced data type and mechanism, but the reason I mentioned it is so that you are aware that it exists. Before we move on, let me at least cover the motivation for its creation and a use case for when you may wish to look into possibly using it. `TypedArray` came into being with ES6 because web applications are getting more advanced all the time, and the client machine now has so much power available to it that writing a client application that processes and manipulates audio and video is a good idea. In order to do this, you need a mechanism to enable your JavaScript code to be able to read and write data to these raw binary streams.

Two examples of something that you may want to build where `TypedArray` would be put to immediate good use are the following:

- Video editing (where you wish to remove segments of unwanted footage)
- Sampling audio (where you change the frequency of a sound byte, maybe creating 11 versions of the original sample to create a chromatic scale, so as to be able to play melodies from the original single sample)

Once again, this is an example of how far JavaScript has come.

Maps

Maps is a data structure that came to JavaScript in ES6. Maps are used for mapping values to values. Moreover, they allow the use of arbitrary values as keys, meaning that you can use integers as keys, or strings, or even objects; however, the use of symbols as keys is not allowed. There are also a few handy methods for performing operations on maps, and you can also iterate over a map. Let's take a look at some code for creating a map and explore some of its common built-in functions. Let's first create our map. We'll create one for mapping learning curves to programming languages, using the new keyword:

```
var mapLangCurve = new Map();
```

And now let's add a few entries to it using the Map's set function:

```
mapLangCurve.set('Python', 'low');
mapLangCurve.set('Java', 'medium');
mapLangCurve.set('C++', 'high');
```

While you can add key-value pairs in Map one line at a time, as we just did, the `set` method is chainable and so we can use this syntax to accomplish the exact same thing, which saves some typing:

```
var mapLangCurve = new Map().
set('Python', 'low').
set('Java', 'medium').
set('C++', 'high');
```

Alternatively, as a third way to declare and initialize our language learning curve Map, we can pass an array of two element arrays to the Map's constructor. Let's assume we had our array of two-element arrays set up like this:

```
var arrLangCurve = [['Python', 'low'],
['Java', 'medium'],
['C++', 'high']];
```

We can then pass that into the constructor like this:

```
var mapLangCurve = new Map(arrLangCurve);
```

These three methods of creating our Map all produce the exact same results. Cool, let's move on and quickly look at some of the common operations that can be done on Map. We can get the Map's size using its `size` property:

```
var langCurveSize = mapLangCurve.size;
```

We can retrieve a key's value using the `get` function:

```
var javaLearningCurve = mapLangCurve.get('Java');
```

We can check for the existence of a key in Map using its `has` function:

```
var blnCurveExists = mapLangCurve.has('C++');
```

We can delete a key and its value using the `delete` function:

```
mapLangCurve.delete('Python');
```

We can clear out a set, removing all its items in one fell swoop, with the `clear` function.

If you're following along in your JavaScript environment, don't try this just yet because we need some data to iterate through it:

```
mapLangCurve.clear();
```

You can iterate over Map very easily in JavaScript using the `for` construct, but we need to know what we would like to iterate over. Do we want to get our map's keys? or its values? Or maybe we'd like to get both. Here is how we iterate over our map's keys:

```
for (let key of mapLangCurve.keys()) {  
    console.log(key);  
}
```

Here is how we iterate over our map's values:

```
for (let value of mapLangCurve.values()) {  
    console.log(value);  
}
```

Here is how we iterate over both our Map's keys and its values:

```
for (let item of mapLangCurve.entries()) {  
    console.log(item[0], item[1]);  
}
```

Most of the time, you'll probably want to have access to your map's keys and values and so you should use the map's `entries` function.

A bit later in this chapter, we'll take a look at a construct that ES6 gave us, called destructuring, which gives us the ability to access keys and values directly.

WeakMap

`WeakMap` is a funny creature and is not inherited from `Map`, although they both are collections of key-value pairs and share some of the same functionality.

The most important difference between `Map` and `WeakMap` is the datatypes that can be used as their keys. With a, we've seen that you can use a variety of data types as its keys, including objects. However, `WeakMap` can only have objects as its keys. This is by design and makes `WeakMap` particularly useful if you only care to have access to the value of the key if the key has not yet been garbage collected. Let that sink in for a moment. I know it sounds like a strange use case, but if you consider that `WeakMap` can help mitigate memory leaks in your JavaScript programs, it may be enough for thinking of how you may be able to make use of `WeakMap` in your code.

The *Weak* part of the data structure's name comes from the fact that `WeakMap` holds the references to its key objects weakly. This is to say that they are candidates for being garbage collected. This fact leads us to this next point. Since our keys may be garbage collected at any point without our involvement, it would not make sense to make them enumerable, and so they are not, and this means that we cannot iterate over the collection.

If you need to iterate over the list of keys or values in your collection, you should use the `Map`. Conversely, if you don't need to iterate over `Map` and just intend to use it as a lookup table, you may want to consider using `WeakMap`.

We will learn about `Set` in the next section.

Set

`Set` is a collection of unique values and can be iterated over in the order in which its elements were added to it. `Set` can contain homogeneous data but each piece of data (that is, element) needs to be unique. If you try to add an existing element to a set, there will be no effect on the set.

Sets have many of the same functionality that maps have. Let's create a `Set` object and zip through a few of its commonly used functions.

To create a set, we call its constructor using the `new` keyword:

```
var setCelestialObjects = new Set();
```

Let's add a few elements (that is, celestial objects) to our Set:

```
setCelestialObjects.add('Earth');
setCelestialObjects.add('Moon');
setCelestialObjects.add('Solar System'); setCelestialObjects.add('Milky Way');
setCelestialObjects.add('Andromeda');
setCelestialObjects.add(['Aries', 'Cassiopeia', 'Orion']);
setCelestialObjects.add(7);
```

OK, the number 7 isn't exactly a celestial object, but I wanted to show you that you can add different types of elements to the same Set. The same thing with our array of constellations: we can add arrays, and any type of object, to our Set.

We can get the size of our Set using the `size` property:

```
var sizeCelestialObjects = setCelestialObjects.size;
```

Don't do this now, but you can clear out Set using its `clear` function:

```
setCelestialObjects.clear();
```

We can delete an element from our Set by passing its value into set's `delete` function:

```
setCelestialObjects.delete('Andromeda');
```

You can iterate over Set using the `for` construct, as we used for our Map:

```
for (let element of setCelestialObjects) {
    console.log(element);
}
```

If you'd like to perform an operation on every element in your Set, you can use set's `forEach` function, which takes a callback as its parameter. For instance, if you had a set of integers in your Set and wanted to square all of them, this is how you can accomplish that:

```
var setIntegers = new Set();
setIntegers.add(1);
setIntegers.add(7);
setIntegers.add(11);
setIntegers.forEach(function(value) {
    console.log(Math.pow(value, 2));
});
```

The preceding code does not change the elements within our Set while it prints the squared values to our console. We can't easily change our elements in place, but we can create a new Set and store our squared elements in there, like this:

```
var setSquaredIntegers = new Set();
setIntegers.forEach(function(value) {
    setSquaredIntegers.add(Math.pow(value, 2));
});
```

We can check for the existence of an element in our Set using the has function:

```
var blnElementExists = setCelestialObjects.has('Moon');
```

If you recall from when we covered maps, the Map object had the following three functions: keys, values, and entries. Sets have these same three functions as well, but their resulting value is quite different. When you call these built-in functions on your Set, you will get a SetIterator object back.

We won't be using SetIterator in this book, but as with when I gave you a use case for TypedArray, I'd like to give you a use case for SetIterator. A Map object and a Set object are different data structures, and you iterate through each of these structures in different ways. If you use iterators, you can build a function that can iterate through these two types of data structures in the same way. In other words, you can pass the objects into your function that iterates over the collections without worrying about their type.

WeakSet

WeakSet is a collect of weakly held objects, where each one must be unique; duplicate objects are not allowed to be added. Recall from our discussion on WeakMap that its keys may be garbage collected from under our feet since their keys can only be objects. Thus, as it was with WeakMap, so it is with WeakSet with respect to iteration: we can't iterate over the collection.

WeakSet has a very small amount of built-in functions, namely add, delete, and has. WeakSet also has a length property, similar to arrays, as opposed to the size property for Map.

Let's take a quick look at the syntax for creating a WeakSet object, and its property and functions.

We can create an empty `WeakSet` object using its constructor:

```
var myWeakSet = new WeakSet();
```

Let's create three empty objects to add to our `WeakSet`, and then add them to it using the `WeakSet` object's `add` function, and then get the number of objects it contains using its `length` property:

```
var objA = {};
var objB = {};
var objC = {};
myWeakSet.add(objA);
myWeakSet.add(objB);
myWeakSet.add(objC);
var lengthMyWeakSet = myWeakSet.length; // lengthMyWeakSet will be set to 3
```

You may be asking, *Wait a minute. You said that the objects must all be unique, and any duplicates will not be inserted into the WeakSet object. All the objects are empty; are they not the same?* True, duplicate objects will be rejected when the insertion operation is tried. However, while our three objects all have the same value (that is, they are all empty), they are in fact three separate and distinct objects.

In JavaScript, as with most other object-oriented languages, it is the object reference (that is, the underlying memory address) and not its contents that determine whether the object is the same as another object or not.

Here is how you can compare two object variables referencing the same object:

```
var blnSameObject = Object.is(objA, objB);
```

`objA` and `objB` each reference empty objects, but these are two different objects; thus, `blnSameObject` will be set to `false`.

If we did the following since the `objB` and `objC` variables point to the same object in memory, the line that tries to add `objC` to `myWeakSet` will have no effect on `myWeakSet` because the underlying object was already contained in the `WeakSet` object:

```
var objA = {};
var objB = {};
var objC = objB; // objB and objC now both point to the same object in memory
myWeakSet.add(objA);
myWeakSet.add(objB);
myWeakSet.add(objC);
var lengthMyWeakSet = myWeakSet.length; // lengthMyWeakSet will be set to 2
```

Classes

Several pages ago, we covered three different ways to create an object in JavaScript. I had also mentioned that we were going to cover how to create an object using the ES6 class syntax later on. Additionally, I had mentioned that JavaScript doesn't have the notion of classes, yet we're covering classes in this section. Let's clear all this up and take a look at how to create classes in JavaScript, and how to create objects from these classes.

For JavaScript releases prior to ES6, the notion of classes did not exist. Rather, whenever you created an object, under the hood the JavaScript runtime would inherit directly from other objects, and not from classes (remember that JavaScript is not a classical object-oriented language; it uses prototypal inheritance). This doesn't make JavaScript *bad*, but it does make it different. In order to bring the same style and semantics from classical object orientation, ES6 brought us the notion of classes. A class is a blueprint for objects, and when we create an object from this blueprint, or template, it is referred to as instantiation. We use a class to instantiate (to bring into existence) one or more objects from it. Let's get into the syntax.

Let's create a `Car` class and give it a constructor, three properties, and three methods:

```
class Car {  
    constructor(make, model) {  
        this.make = make;  
        this.model = model;  
        this.speed = 0;  
    }  
    get speed() {  
        return this._speed;  
    }  
    set speed(value) {  
        this._speed = value;  
    }  
    speedUp() {  
        this.speed += 10;  
    }  
    slowDown() {  
        this.speed -= 10;  
    }  
}
```

I've purposely used the term *methods* here, whereas previously I had always referred to them as *functions*. Since we're now discussing classes and objects, in the classical object-orientation parlance, *method* is of better choice a word than *function*.

The two relationships that you need to remember for any object-oriented language, are the following:

- Objects are instances of their classes
- Objects encapsulate data and methods that manipulate that data

The data represents the state of the object at any moment in time, and the methods represent the behaviors the object has. That's pretty much it.

OK, back to our class example. Our `Car` class has a constructor that takes two parameters: the car's make and model. It also has three instance variables: `make`, `model`, and `speed`. Additionally, it has two methods, `speedUp` and `slowDown`. Lastly, the `speed` instance variable is actually a property; this is because it has an associated getter and setter.

Something to pay attention to is that the setter and getter in our class have an underscore in front of the property name, while the associated instance variable does not. This is important because, without the underscores, the JavaScript runtime would throw an exception (that is, `RangeError: Maximum call stack size exceeded`) when instantiating your `Car` object.

Great! So, how do we create our instance of it (that is, a `Car` object), and how can we call its methods and read its property? Here's the code to help answer these questions.

We create our `Car` object just like any other object, by calling its constructor:

```
var myG6 = new Car('Pontiac', 'G6');
```

Let's read our car's current speed:

```
myG6.speed; // this returns 0, which is the value it was initialized to
```

Oh, man! Zero miles per hour? That's unacceptable. Let's step on the gas pedal! See this:

```
myG6.speedUp(); // this increases our speed by 10 mph
myG6.speedUp(); // this increases our speed by another 10 mph
myG6.speedUp(); // this increases our speed yet again, by 10 mph
myG6.speedUp(); // this increases our speed, one last time, by ... you
guessed it, 10 mph
```

How fast are we going? We'd better check:

```
myG6.speed; // this now returns 40
```

Crap! We just entered a school zone and have to drop down to a maximum speed of 20 mph:

```
myG6.slowDown(); // this decreases our speed by 10 mph  
myG6.slowDown(); // this decreases our speed by another 10 mph
```

Let's check our speed again:

```
myG6.speed; // this now returns 20
```

Whew! OK, we're good at this speed, for now.

To wrap up this section, here are a couple of things to keep in mind about classes in JavaScript:

- Unlike Java or Python, for instance, classes in JavaScript can only have one constructor. Overloading constructors is not supported.
- You can have a super call in your class (used in calling the constructor of a class higher up in the hierarchy), but it must be called prior to using the `this` reference, as in when we assign the `make` and `model` parameters to their respective instance variables.

Interfaces

So far, we have been looking at several new additions to JavaScript that have been made available to us. This section on interfaces, for our purposes, is a TypeScript-specific thing, since JavaScript does not have the notion of interfaces.

Interfaces are like a contract for a class and provide a set of rules that the class must follow. Let's switch gears from building a `Car` class to building an `Animal` class, and, while we're at it, let's have our class implement an interface we'll call `Species`:

```
class Animal implements Species {  
}  
interface Species {  
    name: string;  
    isExtinct: boolean;  
}
```

Our `Animal` class is empty. It doesn't even have a constructor or any instance variables, and that is not a problem for us, since it still serves our purposes to demonstrate how to use interfaces.

Take a look at the `Species` interface for a moment. You'll notice a couple of things:

- It has two public properties. TypeScript has access modifiers, just like Java and C# have, and we'll get to those when we make use of them in later chapters. For now, all you need to know is that the lack of an access modifier on the properties makes the properties public. This is important because since an interface describes the public interface of the class that implements it, its properties must be public.
- The second thing that you'll notice is that we're typing the properties. We're declaring the `name` property as being of type `string`, and the `isExtinct` property as being of type `boolean`. This is one major advantage of TypeScript, as we've previously learned, and is where TypeScript got its name (that is, a typed JavaScript).

We'll see access modifiers in action later in the book. There are three of them:

- **Public:** This is the default modifier and it means that the property or function is visible to all other code
- **Private:** The visibility to a class's properties and functions marked as private are only available to member functions of the class that they're declared in
- **Protected:** This is the same as private, but the class members are also visible to any classes that are inherited from the class that they're declared in

The way in which we marry the class to the interface is by using the `implements` keyword in the class definition, as we have done so in this example. Once we do that, the class must adhere to the interface contract.

So now what? Well, if the `Animal` class doesn't implement the two properties that the `Species` interface says it must, then TypeScript will throw an error during transpilation.

We can also have an interface describe an optional contract, and we can do this by appending a question mark to the end of the property or function. We don't have a function listed in our interface, but we can absolutely have functions as well.

If our interface was an optional contract, it would look like this:

```
interface Species {  
    name?: string;  
    isExtinct?: boolean;  
}
```

Inheritance

We mentioned that class members marked with the protected access modifier are also visible to any classes that are inherited from the class that they're declared in, and so we'd better discuss inheritance real quick.

Inheritance does not mean that the classes that we create will become independently wealthy; that's a whole different kind of inheritance. The kind of inheritance that we're talking about is a little less exciting, but far more useful for our JavaScript code.

A class can inherit from another class. In order for a class to do that, we use the `extends` keyword in the class definition. Let's switch gears once more, this time going from `Animal` to `Employee` (though I've seen some bathrooms and kitchens at a couple of my client locations, and I can tell you that some employees can also be animals). Let's get to the code:

```
class Employee {  
    constructor(name) {  
        this.name = name;  
        this.title = "an employee";  
    }  
    announceThyself() {  
        return `Hi. My name is ${this.name} and I am ${this.title}.`;  
    }  
}  
class Manager extends Employee {  
    constructor(name) {  
        super(name);  
        this.title = "a manager";  
    }  
}
```

Let's create an `Employee` object, and have the employee announce himself:

```
var emp = new Employee("Goofy");  
console.log(emp.announceThyself());  
// Hi. My name is Goofy and I am an employee.
```

Let's create a `Manager` object and have the manager announce himself:

```
var mgr = new Manager("Mickey");  
console.log(mgr.announceThyself());  
// Hi. My name is Mickey and I am a manager.
```

Here's what's going on:

1. We created an `Employee` class.
2. We created a `Manager` class that inherits from `Employee`.
3. The `Manager` class does not have any properties or functions, other than its constructor. However, it inherits the properties (`name` and `title`), and the method (`announceThyself`), from the `Employee` class.
4. The constructor in the `Manager` class calls the constructor in the `Employee` class, passing in the manager's name.
5. The manager's constructor reassigned the value for the `title` property.

This was fairly straightforward but there are two takeaways here for you to remember:

- The inheriting class gets all the class members from the class that it inherits from
- One constructor can call its parent's constructor, and this can continue up the chain, if the parent had a parent, and so on

Destructuring

Destructuring is a super cool and super useful construct that we will use many times throughout this book, and it'll be something that you won't be able to live without in your Angular projects well after you finish this book. In short, destructuring is a JavaScript expression that enables us to easily extract data from objects and collections.

Back when we were looking at `Map` objects, I had mentioned that we would look at an example of destructuring. Here it is.

Assume we have the following object:

```
const author = {  
    firstName: 'Aki',  
    lastName: 'Iskandar',  
    topics: ['Angular', 'JavaScript', 'TypeScript', 'Bootstrap', 'Node'],  
    cities: ['Calgary', 'Cleveland'],  
    publisher: 'Packt'  
}
```

If we wanted to extract `firstName`, `lastName`, and `publisher`, we know exactly how to do that the old-fashioned way (that is, before ES6):

```
const firstName = author.firstName;
const lastName = author.lastName;
const publisher = author.publisher;
```

Well, destructuring (despite it's odd-looking syntax) saves us a lot of keystrokes by giving us the exact same result (new variables with the extracted data) with the following syntax:

```
const {firstName, lastName, publisher} = author;
```

We can easily see that it did its job by writing a variable out to the console:

```
console.log(publisher); // Packt
```

It's pretty handy, and we're going to make good use of it when we write our application together.

Template strings

Template strings are strings that are enclosed within backticks (that is, `).



Note: the backtick character is typically found on the same key on your keyboard as the tilde (that is ~), and is immediately to the left of the number 1 key.

JavaScript always gave us the ability to create strings using double quotes as well as by using single quotes, so what was the motivation for the third type of string creation character? Well, as it turns out, given the proliferation of frontend frameworks, there was a common need to do three things:

- String interpolation
- Multiline strings
- Tagged templates

for-of loop

JavaScript brought us the `forEach` construct for looping through collections. It is a great built-in method to use, but you can't break out of this loop. We also have the `for-in` loop, which is great for objects with keys that are strings, but it has some drawbacks when iterating through arrays.

Enter the new `for-of` loop. It works well for objects, arrays, and maps, and you can break out of it. Here is the syntax, which is identical to that for the `for-in` loop, other than changing `in` to `of`:

```
let myArray = [5, 10, 15, 20, 25];
for (var item of myArray) {
  console.log(item);
}
```

Decorators

Decorators are also a TypeScript thing. Decorators in TypeScript decorate functions and classes, just as they do in some other languages, such as Python and Java.

We won't spend too much time here because we won't be writing our own decorators for the application we're going to build together, but since Angular makes use of decorators quite a bit, I wanted to at least give you an idea of what they are used for. We'll also look at a quick example of how to create one and how to use it, but we'll fly through it quickly.

Decorators are a way to add functionality to a function or a class (typically a class), by annotating the class with the decorator. The decorator is just a function, although it has some strange looking syntax at first glance. Let's look at some code:

```
function iq(target) {
  Object.defineProperty(target.prototype, 'iq', {value: () => "Genius"})
}
@iq
class Person {
}
let readerOfMyBook = new Person();
console.log(readerOfMyBook.iq()); // prints out "Genius" to the console
```

This is intermediate to advanced level TypeScript, and an entire chapter can be written on decorators. We don't have the luxury to cover them in detail here, but the takeaway is that they are simply functions that add functionality to functions or classes, and to do that you only need to annotate the function or class with the name of the decorator (that is, `@NameOfDecorator`).

Promises

We will cover promises in greater detail when we make use of them in Chapter 12, *Integrating Backend Data Services*, and so I'll defer the code until that time. The reason is that showing a really good real-world example of a promise, so that it's not contrived, takes quite a bit of code, since there needs to be asynchronous code that is called. So, I promise to have real-world promises later on in the book. However, we can at least look at a definition so that you know what they are.

When you call a function that may take a long time to return a result or to complete its task, and you don't want to delay the execution of your program, you can call that function asynchronously. This means that your code will continue on to the next line after it calls the function on the previous line asynchronously. If you don't call it asynchronously, your program's execution will stop and wait for the function you last called to return from what it was doing, such as reading a bunch of records from a database.

There are a few different ways to call a function asynchronously. The most common way to call a function asynchronously is to use callbacks. A callback is a function that you pass to the function that you call asynchronously so it can then call that function when it has completed its work. That's how callbacks got their name; the function you called calls you back when it's done.

Promises are another mechanism we can use to program asynchronously. Although Promises made things a little more manageable, writing good asynchronous code in JavaScript is often still notoriously difficult. Because of this fact, people started writing JavaScript libraries to try and make asynchronous code easier to write. There are several out there. One library that saved my sanity is called Async.

All this being said, I still have not given you a definition of Promises, so here it is: a Promise is a proxy for a value that is not yet known; it's like a placeholder for the value that will eventually come back from a function that was called asynchronously. This construct allows asynchronous functions to immediately return a value as if it was a synchronous method. The initial value that is returned is a Promise, and the Promise will eventually be replaced by the value that comes back from the called function once it has completed its work.

I know this may be a lot to get your head around, but when we write our code in Chapter 12, *Integrating Backend Data Services*, you will understand Promises. That's a promise, pun intended.

Modules

Prior to ES6, JavaScript did not have the notion of modules. Modules are simple code files that can be loaded into the other code so that the functions within the module that is being loaded are made available to the code that is importing the module. Modules can load modules. Modules lead to modular code, and that is a good thing. Rather than write a monolithic bunch of code in one file, you can split it up into logical units and have that code live in more than one file. This leads to code reuse, namespacing, and maintainability.

While JavaScript didn't have modules, we were still able to accomplish the same thing, to a degree. We can load script files with script tags before calling their functions in our web pages. However, what about JavaScript running on the server side, or another environment outside of web pages? Without modules, writing non-monolithic applications becomes difficult.

Let's move on to the code.

Assume we have a file named `alphafunctions.js` that has the following code in it:

```
function alpha1() {
    console.log("Alpha 1 was called");
}
function alpha2() {
    console.log("Alpha 2 was called");
}
export {alpha1, alpha2};
```

The `export` keyword is used to mark which functions can be exported and thus imported into other modules.

Let's now assume we have this file, `main.js`, with the following code:

```
import {alpha1, alpha2} from './alphafunctions';
alpha1(); // "Alpha 1 was called" is written to the console
```

Default exports

Let's assume that we always want to import our `alpha1` function into other modules, or at least more often than not. We can add the keyword's `export default` before the keyword `function`. So, when we import it, we no longer need the curly braces around the function name. Let's see this in the code.

See `alphafunctions.js`:

```
export default function alpha1() {
    console.log("Alpha 1 was called");
}
function alpha2() {
    console.log("Alpha 2 was called");
}
export {alpha1, alpha2};
```

See `main.js`:

```
import alpha1, {alpha2} from './alphafunctions';
```

While this isn't an earth-shattering difference, the terms default exports do come up in conversation and in code snippets on blog posts, and so on, so I wanted to make sure we at least took a quick look so that you understood why the curly braces were sometimes there, and other times not. When you use JavaScript libraries, you'll see this in the documentation and code examples as well. So, now you know.

Summary

In this chapter, we have covered some of the history around JavaScript, and specifically the series of fortunate events around JavaScript's ecosystem that have cemented the language as the most important programming language in recent history. We can now not only write client-side code for execution in the browser, but we can write JavaScript code that runs on the server. As if this wasn't a compelling enough reason to use more JavaScript than what you may have used in the past, you can also use JavaScript for native mobile development, and for creating desktop applications as well. It's quite an exciting story!

We then took a low fly-by look at the additions to JavaScript that were added with the release of ES6. These additions were quite substantial, especially since JavaScript had remained largely unchanged for over a decade, and have really strengthened the language as a result. We also enumerated some of the benefits that TypeScript brings to the table. Remember that you can view TypeScript as being a superset of JavaScript, and you can define it as ES6 plus optional typing.

Microsoft's contribution of TypeScript to JavaScript developers was one of the most important things the company has contributed to the open source world in a long time. Angular itself was written in TypeScript, due to the advantages TypeScript has over pure JavaScript, and so when building Angular applications, it's best to use TypeScript to write them. We've remembered that JavaScript is the only language that can be executed in the browser, but fortunately, TypeScript ships with a transpiler to turn our TypeScript code into pure JavaScript code.

As mentioned in the roadmap, chapter 3 *Bootstrap – Responsive Grid Layout and Components*, we have a similar goal. In this chapter we will do a crash course on SASS, which is what we'll be using to style our Angular components instead of using CSS. We'll also cover enough of Bootstrap to get you comfortable with using the venerable CSS framework to lay out our web pages for ListingCarousel, the web application we'll be building together. You'll gain enough knowledge to immediately apply these skills to virtually any web application project you may currently have or soon be starting on in the future.

3

Bootstrap - Grid Layout and Components

Hey—you made it to Chapter 3, *Bootstrap – Grid Layout and Components*. Awesome! Next on the agenda is to pour yourself a cup, or glass, of your favorite beverage, because this is a big and important chapter—and so you're going to want to be awake for it. However, it's not going to be all work and no play, because this chapter is where we're going to start building our example application together and we should be able to have some fun with it. Along the way, we'll also be covering quite a bit of material in various areas. Here's the list of things that we're going to cover:

- We're going to formally introduce our example application, called Listing Carousel, and this includes some suggestions for how you may leverage this application into something else that you may prefer doing—either alongside this book (instead of building Listing Carousel), or after finishing the book, if you'll first be building Listing Carousel along with me.
- We're going to cover our game plan for how we're going to incrementally build our application throughout the book, and you will also see that we have a few alternative technologies to choose from for building Listing Carousel—or for an application of your choice that you may be inspired to build by the end of the book.
- We're also going to take a look at Sass, which is a technology for making writing your CSS for your projects a little easier, and even more organized.
- We'll definitely be looking at Bootstrap—at both of its major parts: its responsive grid, and a few of its components.

- In Chapter 1, *Quick Start*, we took a sneak peek at some of the wireframes that will make up our example application. Well, this chapter is where we're going to write the HTML code, leveraging Bootstrap, to bring the wireframes to life.
- As bonus material, we'll also be looking at the process in which software projects go from inception to realization, using a real-life case study, Listing Carousel. This includes project phases, such as analysis, requirement gathering, use cases diagrams, wireframing, and implementation.

By the end of this chapter, our web pages will be hardcoded, and won't have any Angular code in them whatsoever. As we progress through the book, we will start to slowly bring them to life by gradually converting our Bootstrap application into a fully fledged Angular application by adding routing, Angular components, templates, data, and more.

A word about what this chapter is not

This chapter covers a lot of ground, including Sass, and Bootstrap's responsive grid and several of its components. However, this chapter is not comprehensive coverage of all you need to know about either of these things. The reasoning for this is simple—not only are there books devoted to Bootstrap, but Bootstraps's website is the ideal place for where to look up documentation on Bootstrap. Duplicating their documentation would not have been a good use of this book's pages, nor would it have been a good use of your time and hard-earned money. Instead, the smarter thing to do is to introduce Bootstrap's grid and components in a practical way—such as by building an application together in this book—and to just refer you to the official documentation as needed.

By the way, the same thing applies to Chapter 5, *Flex-Layout – Angular's Responsive Layout Engine*, Chapter 8, *Working with NG Bootstrap*, and Chapter 9, *Working with Angular Material*, because each of these technologies has their own official documentation.

My job is to do the following:

- Introduce these technologies to you (and point you to their official documentation)
- Demonstrate how they can be applied in a practical, interesting, and, hopefully, engaging way
- Encourage you to work through the entire book, so you can be well on your way to becoming an Angular web development guru

By the way, Angular, of course, also has its own official documentation, but there's so much to it that it can be intimidating to even get started. In my experience, a far more interesting way to learn a new technology is by following a tutorial—and that is exactly what this book is—a comprehensive tutorial for building an application, with the added explanations and some bonus material sprinkled in throughout the chapters in the appropriate sections. If I did my job well, you should be able to build almost any application you are likely to need (or want) to build with Angular. That's the goal.

Let's now take a look at Listing Carousel, which is the example application we'll be building together.

Our example application

Listing Carousel, our example application for this book, is a real-life online application that provides realtors (that is, professional real-estate salespeople) with an opportunity to share their listings to their contacts on social media in an engaging and informative way. One of my companies owns and operates it.

The reason I chose this application is not so you can steal my code and try to compete with me (which would be totally uncool and not recommended), but rather because with some tweaks here and there, you can turn the application into your very own online service if you wanted to. For example, you can easily turn this application into a classified application (such as Craigslist, or Kijiji) by just adding search capability, or even into a dating/matchmaking site by adding search, and just a bit more code. Or, if you love food, how about turning it into a restaurant site? Sign up restaurants to list their menus in the carousel—one meal or appetizer per slide—and then the restaurant owners can share their menus with their social media circles. Or, if you like having new ways to share photo albums, you can turn the application into something like that. One idea I had a while ago was to create a site where people can showcase their portfolios (such as for artists, architects, and photographers)—feel free to build something like that and run with it. The options are truly endless. The point is, I wanted to come up with a fun application for the book—one that would give you some motivation to work through the entire book. Why? Simple—because I know that this book would not be as valuable as it can be to you if all you were to do is to read it. So, commit to dive into the code with me and build something that you would enjoy. Who knows, maybe you will come up with a great idea for a profitable online business! My goal is to make the time you invest in working through this book as time well spent and, if I succeed, you can then give this book a five-star review (wink). Does this sound good?

Game plan

There's a step-by-step game plan for using Listing Carousel as our focal point for our discussions as we cover the material in this book together. Though this book was not explicitly broken up into parts (that is, groupings of chapters), we can loosely group them now by partitioning the work we need to do for building our application into three main stages. Follow along with me, and this will all make sense—giving us a way to marry the material (that is, the book's chapters) to the application we'll be building together, as well as to give ourselves a target for what we're shooting for.

It's nice to know where you're going before you start driving and to be able to recognize where you are at all times. Having a roadmap/game plan like that makes the entire process more enjoyable, thereby maximizing the chances that you'll work through the book instead of just using it to look things up here and there. This book wasn't designed to be like a cookbook. Instead, it was designed to teach you how to cook. You will learn to cook by fire (pun intended), by preparing a meal of the right complexity—which demands a certain level of knowledge and skill to cook it properly. There are four main benefits of this book:

- It gives you all the ingredients and even substitutes ingredients (that is, choices) that you need to prepare the meal.
- It gives you the knowledge and teaches you the process and skills required of the chef in order to cook the meal.
- It does these things in a methodological way so you learn it effectively and as efficiently as possible.
- The meal selection is a dish representative of the complexity of the majority of dishes you're likely to need to cook up. In other words, if you learn how to cook this meal (that is, our example application), you should be confident in being able to prepare any meal that you will be asked to prepare.

Cooking analogies aside, this book's promise is to teach you how to build a practical application using Angular through a methodological process. After all, if you think about it, that's why you bought this book—isn't it? Otherwise, you could have just tried to Google things here and there, hoping to be able to eventually piece everything together. That's not fun, nor is it an intelligent way to go about learning Angular. When you want to learn something new, or take rudimentary skills to the next level—in anything, not just Angular—you need to have a target in mind, and a roadmap/game plan to get there. Well, we know our target, which is to build Listing Carousel, learning Angular, and a bunch of other goodies, along the way. So let's now take a look at our game plan.

In phase 1 of building our application, we need to decide what to build, what features it will have, and what it will look like. Once we have that all scoped out and wireframed, the next step is to build out the skeleton for our application. By the end of this stage, our application is likely to just be hardcoded—being nothing more than some HTML and CSS. Our only components will be the ones we select to use from Bootstrap's library of components. You can think of this as our application having skin and bones, but not yet having guts or a beating heart.

In phase 2 of building our application, well, you guessed it, we're going to start giving our skin and bones application some guts! Specifically, it will be Angular guts! Remember, in the first phase, we're not even using Angular at all—not a single ounce of it—which is intentional. Though our application will surely be more lively by the end of this second phase than what it will be in its first phase, it'll behave more like a robot—very mechanical. If you remember the movie, *The Wizard of Oz*, it will be like the Tin Man—very much alive, but with no heart. This second phase (that is, giving our application some guts) will be comprised of Chapter 4, *Routing*, Chapter 7, *Templates, Directives, and Pipes*, and Chapter 6, *Building Angular Components*.

Last, but not least, in phase 3 of building our application, we'll finally be giving our Tin Man a heart! Yeah! OK—what gives our application a heart? The answer is data and APIs! The data is like blood for our application, and the APIs are like the heart—taking data in and pushing data back out. I bet you never thought of web applications in this way, and, from now on, you won't be able to think of them in any other way (smiling). This third phase will comprise Chapter 10, *Working with Forms*, Chapter 12, *Integrating Backend Data Services*, Chapter 11, *Dependency Injection and Services*, and Chapter 14, *Advanced Angular Topics*.

Chapter 13, *Unit Testing*, and Chapter 15, *Deploying Angular Applications*, are not really parts of any of the phases *per se*, but they play a very important supporting role. In these two chapters, we will learn how to test the code we write for our application, and how to deploy it in a couple of different ways.

That's our big picture look at our game plan. Let's zoom in a little, and take a look at our five-step game plan for our first phase of building our application, and we'll then be on our way to our first technology topic, SASS.

- **Step 1:** In this chapter, we're going to look at Bootstrap's responsive grid, as well as several of Bootstrap's components:
 - I'll be explaining how Bootstrap's grid works and can help us lay out our web pages.
 - I'll cover the Bootstrap components that we'll be using on our pages as we build out the pages—and we'll be using our wireframes to guide us, in conjunction with Bootstrap's grid.
- **Step 2:** In Chapter 5, *Flex-Layout – Angular's Powerful Responsive Layout Engine*, we're going to replace Bootstrap's grid system with Flex-layout. We'll only be doing this with a couple of web pages, leaving all the others with the Bootstrap grid. There are two reasons why we'll be doing this:
 - To show you that there are always alternate options available—and you can often mix and match these alternatives.
 - Being able to mix and match provides us with a path to replace one technology with another without the need to do it all in one shot. We don't want to keep rebuilding things in their entirety—we just want to redo enough of a couple of pieces of what we built originally in order to learn how to apply that specific alternate technology.
- **Step 3:** In Chapter 6, *Building Angular Components*, we'll be looking at how we can build our own components for use in our web pages. Since we are in control of the HTML and CSS when creating our components, we can leverage Bootstrap's components for the look and feel when creating our own Angular components. Note: Chapter 7, *Templates, Directives, and Pipes*, is a part of this as well, since these two chapters go together.
- **Step 4:** In Chapter 8, *Working with NG Bootstrap*, we will discover that there are ready-made *Angular-ready* Bootstrap components. Just as we will replace Bootstrap's grid with Flex-layout for a couple of our pages, we will do the same thing with components—that is, to replace a couple of Bootstrap components with components from the NG Bootstrap project. Our motivation for doing this is to realize that there are many different third-party components that we can readily use for our Angular applications—including ones that are based on Bootstrap's components.

- **Step 5:** In Chapter 9, *Working with Angular Material*, we will once again replace a couple of our Bootstrap components, but this time, they will not have any relation to Bootstrap components. The Angular Material project has beautifully designed components that are tailor-made for use in Angular applications and we'll learn how to incorporate a couple of those into our application.

Again, the important thing to note here is that we have choices in terms of technologies for both laying out our web pages, and as to which components we use—including creating our own custom components—when building our Angular applications. Furthermore, as you'll see in Chapter 12, *Integrating Backend Data Services*, you have virtually unlimited choices as to server-side and database technology stacks. And, in Chapter 14, *Advanced Angular Topics*, we will explore a couple of third-party authentication APIs we may want to leverage for our application instead of writing our own from scratch.

Yup! We have a ton of interesting stuff to cover in this book together. That said, let's focus on first things first, and get started with the goodies this chapter has to offer: Sass, Bootstrap, the typical evolution of software projects (that is, inception to realization), and building out our pages for Listing Carousel using Bootstrap (that is, Phase 1 of building our application). I'll provide a similar game plan for phase 2 of building our application at the start of Chapter 7, *Templates, Directives, and Pipes*, and one final game plan for phase 3 of building our application at the start of Chapter 12, *Integrating Backend Data Services*.

I know that was a lot of ground to cover, but reviewing our game plan was an important thing for us to do—it's always helpful to know where we are and where we're going. Now, let's pick up the pace and rip through the Sass crash course before our coverage of Bootstrap.

Sass crash course

As is the case for most technologies, including all the ones mentioned in this book, such as ES6, Bootstrap, Node, and MongoDB, entire books can be written about them. Sass is no different. The goal of this crash course is not to turn you into a Sass expert, nor is it to regurgitate Sass's official documentation. Due to space constraints, the goal of the crash course is to merely introduce Sass to you, and to motivate you to explore it further on your own, either after you complete this book, or in parallel with it, because Sass is a really cool technology.

The Bootstrap team has adopted Sass for the Bootstrap project, and other technologies (such as Compass) are built on top of it. Strictly speaking, you don't have to know how to use Sass in order to write Angular applications—or even to work through this book—but it's a worthwhile technology to learn, and so I encourage you to take a closer look on your own. Let's cover some Sass basics together now.

What is Sass?

- Sass is an acronym for Syntactically Awesome StyleSheets—but, of course, there's more sassiness to Sass than the acronym! Sass is an extension to CSS, which gives us additional power and flexibility when writing our CSS for our web applications. Sass, when compiled, generates well-formatted CSS for us. Specifically, the additions to CSS that Sass brings to the table include things such as nested rules, variables, conditional logic, mathematics, mixins, and more. Additionally, Sass makes it easier to maintain and organize the style sheets in our web projects. We'll be taking a look at many of these things in this crash course.
- Sass is compatible with all versions of CSS, not just CSS3 and CSS4.
- Thanks to the Angular CLI, Sass fits nicely into our Angular applications because the CLI compiles the SASS in our components for us by default.
- Sass's official website can be found here: <https://sass-lang.com/>.

Compass framework

Compass is a CSS authoring framework that is built on top of Sass, providing some neat additions, and will also compile your Sass to CSS. A compass is an option for you if you're working on non-Angular projects (remember, the Angular CLI takes care of compiling Sass to CSS for us for our Angular projects). We won't be covering Compass in this book, but I wanted to at least bring the technology to your attention because I know that Angular is not the only technology you'll be using as a web developer—however, as web developers, we can't avoid using CSS!

The takeaway point here is that you can simply use the Angular CLI for Sass for Angular projects, but do take a look at leveraging Compass for your non-Angular projects, especially if your project tends to be CSS heavy.

Large companies use Compass. Two of them that I know of, and whose online services I utilize on a daily bases, are LinkedIn (<https://www.linkedin.com/>), the world's largest employment-oriented social networking service, and Heroku (<https://www.heroku.com>), a super popular cloud application platform.

You can learn all you want to about Compass at their official website, which can be found here: <http://compass-style.org/>. Another nice online reference that provides tutorials on Sass and Compass is named *The Sass Way*, and can be found here: <http://www.thesassway.com/>.

Two SASS styles

Sass has two flavors of syntax: the older syntax that relies on indentation, and the newer syntax that uses curly braces as opposed to indentation. Another difference between the two syntactical styles is that the old-style does not require semicolons at the end of the lines, where as the new style does require them. The file extensions between these two styles also differ—the older style's file extension is `.sass`, and the current style's file extension is `.scss`.

Let's now look at a quick example of each one's CSS syntax. The first code block is the older style (`.sass`), and the second code block produces the same effect in the newer syntactical style (`.scss`). We'll be using the new style throughout the book.

The sample code given here is for writing `.sass` syntax:

```
$blue: #0000ff  
$margin: 20px  
  
.content-navigation  
  border-color: $blue  
  color: lighten($blue, 10%)  
  
.border  
  padding: $margin / 2  
  margin: $margin / 2  
  border-color: $blue
```

The sample code given here is for writing `.scss` syntax:

```
$blue: #0000ff;  
$margin: 16px;  
  
.content-navigation {  
  border-color: $blue;
```

```
    color: lighten($blue, 10%);  
}  
  
.border {  
  padding: $margin / 2;  
  margin: $margin / 2;  
  border-color: $blue;  
}
```

The main difference between the two syntactical styles is that the older style aims to be terse, while the newer style aims to be more familiar to developers used to traditional CSS syntax.

In the preceding code blocks, you may have noticed `$blue` and `$margin`. These are not CSS items, but rather they are examples of variables. You've probably also noticed division operators. Variables and mathematical evaluation are just a couple of things that can be in your Sass code. We'll see these and more Sass features a bit later in the sections that follow.

Regardless of which syntax you use—old or new—the compiled result is the same. If you were to take either of the preceding code blocks and run them through an online compiler, such as *Sass Meister* (I'll mention this tool shortly as well), the resultant CSS will be the following:

```
.content-navigation {  
  border-color: #0000ff;  
  color: #3333ff;  
}  
  
.border {  
  padding: 10px;  
  margin: 10px;  
  border-color: #0000ff;  
}
```

Installing Ruby

Sass is written in Ruby, and so we need Ruby installed on our computers. To download and install the latest version navigate to Ruby's official site: <https://www.ruby-lang.org/en/downloads/>. To find out if you already have Ruby installed on your machine, run this command on your command line or Terminal: `$ ruby -v`.

If Ruby is installed, the output will display the version number. For example, the output in my Terminal when I execute `$ ruby -v` is `ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]`. Any version from 2.3 onward is more than good enough for our purposes.

Installing Sass

Once you have Ruby installed, installing Sass is a breeze. Head on over to <https://sass-lang.com/install> and follow the directions.

Just like you can get the version of Ruby by running `$ ruby -v` at your Terminal or command line, you can do the same thing with Sass. Execute this following command, `$ sass -v`, to see what version of Sass you have on your system at your Terminal or command line. The output on my system shows the following:

```
Sass 3.5.5 (Bleeding Edge) .
```

Online tools for Sass

There are several online tools that we can use to compile our Sass files down to CSS files. One of the ones I like is named *Sass Meister*, and you can access it here: <http://www.sassmeister.com>.

I like it because it is super easy to use and provides decent help when there is a problem with your Sass syntax. It also supports both Sass syntaxes, the older style, and the newer style, and it also allows you to choose from a few different Sass compilers. You can find these settings under the **Options** menu option at the top of the window panes.

To use the tool, simply write your Sass code in the left pane, and the compiled CSS will appear in the right pane. Pay attention to the selected options so that the ones that are activated are the ones you want.

Offline tools for Sass

Just as with online tools, we have several options that we can choose from to compile our Sass files down to CSS files using offline tools. I've used Koala because it's easy to use, is cross-platform, and is free. You can download Koala for free from the project's website: <http://koala-app.com/>.

Koala gives you the ability to work with more than just Sass. You can also use Koala to compile and/or compress Less, JavaScript, CoffeeScript, and you can even use the Compass framework with it.

The best way to learn how to use Koala is to read the official documentation, which can be found at <https://github.com/oklai/koala/wiki#docs>. But if you'll just be using Koala to compile your Sass files (at least for now), let me quickly outline the steps for you here to save you from needing to jump back and forth between the book and the online docs.

All you need to do is to create a web project using any text editor of your choice—such as Sublime Text or Visual Studio Code—and create a CSS folder as well as a Sass folder in your project's root folder. You don't need a completed project, of course—all you need is the very basic folder structure. Once the project structure is created, you can open Koala to get started, using it to compile your Sass files for you. Here are the basic steps:

1. Create an empty project structure, which, as a minimum, has the following:
 - A root folder with an empty `index.html` page in it
 - A CSS folder in the root folder with an empty `styles.css` file in it
 - A Sass folder in the root folder with an empty `style.scss` file in it
2. Open the Koala application.
3. Click the large plus sign (+) in the top-left corner, navigate to your project's root folder, and select it. At this point, Koala will find your `styles.scss` and `styles.css` files.
4. Right-click on your `styles.scss` file in Koala's right-hand pane, select **Set Output Path**, and then navigate to, and select, your `styles.css` file in your file explorer

Observing the preceding steps is all you need to do to set Koala up to compile your Sass files for you. The output of the compilation will be inserted into your `styles.css` file.

Sass features and syntax

Let's now take a look at some of Sass's features that you are most likely to use in your applications. We won't be using all of these in our example application, but I wanted to show you some of the cool stuff that Sass has to offer.

Nesting

Using Sass, you can nest CSS rules within each other. Sass is not only easier to read, but it helps to avoid a lot of duplication of the CSS selectors. This is especially true for highly-nested CSS rules. Let's look at a simple example:

```
/* Here is some basic Sass that uses nesting */

#outer-frame p {
  color: #ccc;
  width: 90%;
  .danger-box {
    background-color: #ff0000;
    color: #fff;
  }
}
```

The preceding Sass code will be compiled and the equivalent CSS code will be generated:

```
/* This is the CSS that the above Sass is compiled to */

#outer-frame p {
  color: #ccc;
  width: 90%;
}
#outer-frame p .danger-box {
  background-color: #ff0000;
  color: #fff;
}
```

Variables

Sass variables are just like you would expect them to be: they store information that you'd like to reuse throughout your style sheet. This saves time and annoying errors. And just like global variables in other languages, you only need to define them in one place—so if they ever need to change, you just have to change the variable in one place, rather than change all of the occurrences in your CSS styles.

You can store almost anything. Here is an example where we store the font information and a font color:

```
/* Here is the Sass code defining the variables */

$font-stack: Helvetica, sans-serif;
$primary-color: #333;
body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

The preceding Sass code will be compiled and the equivalent CSS code will be generated:

```
/* Here is the CSS that the above Sass is compiled to */

body {
  font: Helvetica, sans-serif;
  color: #ccc;
}
```

Mathematical operations

Since Sass compiles down to CSS, you can have it do mathematical calculations for you instead of doing them yourself. You can also have the math be run on variables, as opposed to hardcoded numbers as in the following example, which, of course, is super handy to be able to do:

```
/* Here is Sass that has some math in it */

.main-container { width: 100%; }
article {
  float: right;
  width: 700px / 960px * 100%;
}
```

The preceding Sass code will be compiled and the equivalent CSS code will be generated:

```
/* Here is the CSS that the above Sass is compiled to */

.main-container {
  width: 100%;
}
article {
  float: right;
  width: 72.91667%;
}
```

Imports

Sass enables you to import one style sheet into another one using the `@import` directive. This does as it sounds, and is very straightforward. Let's look at an example. In the following three code listings, the first one is the base style (`base.scss`) sheet that is applied to the entire site, and the second is the style sheet that is used for the report pages (`reports.scss`). The third is the resultant CSS style sheet we would get when the reports style sheet imports the base style sheet during Sass compilation. Note that the file extension is not required when using the `@import` directive in Sass:

```
/* base.scss */
body {
  margin: 10px;
  padding: 10px;
  font: Helvetica, sans-serif;
  color: #333;
  background-color: #eee;
}

/* reports.scss */
@import 'base';
p {
  margin: 5px;
  padding: 5px;
  color: #0000CD;
  background-color: #EEE8AA;
}
```

The preceding Sass code will be compiled and the equivalent CSS code will be generated:

```
body {
  margin: 10px;
  padding: 10px;
  font: Helvetica, sans-serif;
  color: #333;
  background-color: #eee;
}
p {
  margin: 5px;
  padding: 5px;
  color: #0000CD;
  background-color: #EEE8AA;
}
```

Extend

Using `@extend` lets you share a set of CSS properties from one selector to another. One selector can extend (that is, inherit) from another one using the `@extend` Sass directive. The following example shows a common set of style attributes for a set of three related styles—active, inactive, and terminated:

```
%common-status-styles {  
  width: 200px;  
  height: 75px;  
  padding: 10px;  
  color: #333;  
}  
  
.active {  
  @extend %common-status-styles;  
  background-color: green;  
  border-color: #001a00;  
}  
  
.inactive {  
  @extend %common-status-styles;  
  background-color: yellow;  
  border-color: #999900;  
}  
  
.terminated {  
  @extend %common-status-styles;  
  background-color: pink;  
  border-color: #ff5a77;  
}
```

When the preceding Sass code gets compiled, it turns into the following CSS:

```
.active, .inactive, .terminated {  
  width: 200px;  
  height: 75px;  
  padding: 10px;  
  color: #333;  
}  
  
.active {  
  background-color: green;  
  border-color: #001a00;  
}  
  
.inactive {  
  background-color: yellow;
```

```
border-color: #999900;  
}  
  
.terminated {  
background-color: pink;  
border-color: #ff5a77;  
}
```

Mixins

Mixins are like named templates. They are Sass's way to allowing you of group CSS or Sass declarations (that is, CSS styles) together and give it a name. This way, you can include these declarations in other CSS classes as needed, without copying and pasting—which causes a bit of a mess should anything need to change later on. In a sense, they are also like variables because you only need to change something in one place (that is, in the mixin itself), but they are more powerful than variables, which is why I mentioned that they are like templates. In fact, mixins can even be parameterized using variables. Let's look at an example, and the preceding description should become clear as to what I mean when I say that mixins are like templates.

Here is an example of styling a dropdown that I like to use in my websites. We'll parameterize the width so that we can create different sizes of dropdowns. Note the use of the `@mixin` directive:

```
@mixin custom-dropdown($dropdown-width) {  
-webkit-appearance: button;  
-webkit-border-radius: 2px;  
-webkit-box-shadow: 0px 1px 3px rgba(0, 0, 0, 0.1);  
-webkit-padding-end: 20px;  
-webkit-padding-start: 2px;  
-webkit-user-select: none;  
background-image:  
url(https://www.maxfusioncloud.com/static/img/15xvbd5.png),  
-webkit-linear-gradient(#FAFAFA, #F4F4F4 40%, #E5E5E5);  
background-position: 97% center;  
background-repeat: no-repeat;  
border: 1px solid #AAA;  
color: #555;  
font-size: 10pt;  
margin: 0px;  
overflow: hidden;  
padding: 5px 12px 6px 6px;  
text-overflow: ellipsis;  
white-space: nowrap;  
width: $dropdown-width;  
}
```

And here is how we can use the mixin (note the use of the @include directive):

```
.small-dropdown { @include custom-dropdown(75px); }
.medium-dropdown { @include custom-dropdown(115px); }
.large-dropdown { @include custom-dropdown(155px); }
```

This will compile to the following CSS:

```
.small-dropdown {
  -webkit-appearance: button;
  -webkit-border-radius: 2px;
  -webkit-box-shadow: 0px 1px 3px rgba(0, 0, 0, 0.1);
  -webkit-padding-end: 20px;
  -webkit-padding-start: 2px;
  -webkit-user-select: none;
  background-image:
    url('https://www.maxfusioncloud.com/static/img/15xvbd5.png'),
    -webkit-linear-gradient(#FAFAFA, #F4F4F4 40%, #E5E5E5);
  background-position: 97% center;
  background-repeat: no-repeat;
  border: 1px solid #AAA;
  color: #555;
  font-size: 10pt;
  margin: 0px;
  overflow: hidden;
  padding: 5px 12px 6px 6px;
  text-overflow: ellipsis;
  white-space: nowrap;
  width: 75px;
}

.medium-dropdown {
  -webkit-appearance: button;
  -webkit-border-radius: 2px;
  -webkit-box-shadow: 0px 1px 3px rgba(0, 0, 0, 0.1);
  -webkit-padding-end: 20px;
  -webkit-padding-start: 2px;
  -webkit-user-select: none;
  background-image:
    url('https://www.maxfusioncloud.com/static/img/15xvbd5.png'),
    -webkit-linear-gradient(#FAFAFA, #F4F4F4 40%, #E5E5E5);
  background-position: 97% center;
  background-repeat: no-repeat;
  border: 1px solid #AAA;
  color: #555;
  font-size: 10pt;
  margin: 0px;
  overflow: hidden;
```

```
padding: 5px 12px 6px 6px;
text-overflow: ellipsis;
white-space: nowrap;
width: 115px;
}

.large-dropdown {
-webkit-appearance: button;
-webkit-border-radius: 2px;
-webkit-box-shadow: 0px 1px 3px rgba(0, 0, 0, 0.1);
-webkit-padding-end: 20px;
-webkit-padding-start: 2px;
-webkit-user-select: none;
background-image:
url(https://www.maxfusioncloud.com/static/img/15xvbd5.png) ,
-webkit-linear-gradient(#FAFAFA, #F4F4F4 40%, #E5E5E5);
background-position: 97% center;
background-repeat: no-repeat;
border: 1px solid #AAA;
color: #555;
font-size: 10pt;
margin: 0px;
overflow: hidden;
padding: 5px 12px 6px 6px;
text-overflow: ellipsis;
white-space: nowrap;
width: 155px;
}
```

By now, you can see how cool Sass is and how much time it can save you, as well as how you can use it to avoid code duplication and avoid making the silly cut and paste mistakes.

And as if all this wasn't cool enough, Sass gives you a lot of power with its built-in functions. There are a ton of them, which is what gives Sass so much power and utility. You can take a look at them here: <http://sass-lang.com/documentation/Sass/Script/Functions.html>. We'll only be covering one, just to show you an example of how to use a function in the *Built-in Functions* section that follows.

Placeholders

Placeholders are intended to be used with the `@extend` directive. Rulesets that do make use of placeholders, but which do not use the `@extend` directive, will not get rendered to CSS. A valid use case of using placeholders is if you are writing a Sass library for code reuse. You can write a Sass file that has placeholders, and which is meant to be included in another Sass file that you or someone else writes. If a ruleset in the Sass file imports the other Sass file that acts as a library, and the ruleset extends the placeholder in your library, it will be rendered to the CSS file when the Sass files are compiled. And, if no rulesets extend the placeholder, the placeholder will not be rendered/printed to the CSS file.

Let's take a look at an example. Note that the placeholder is prefixed by a percent sign (%):

```
%warning-placeholder {  
  color: red;  
  font-weight: bold;  
  font-size: 1.5em;  
}  
  
.warning {  
  @extend %warning-placeholder;  
}
```

The preceding Sass code compiles down to the following CSS:

```
.warning {  
  color: red;  
  font-weight: bold;  
  font-size: 1.5em;  
}
```

Built-in functions

When we covered Sass's extend feature, the border colors for each class were 20% darker than their corresponding background colors. In order to find a color that is 20% darker than another, you have to do some tedious math—and if you decide to later change that percentage, it'll require more tedious math. Fortunately, Sass has built-in functions for us to do all sorts of things, including darkening and lightening colors and a whole lot more.

Now, let's revisit the Sass code we had seen in the previous *Extend* section, and this time write it more flexibly using variables and the built-in `darken` function in order to have Sass do the math for us. This way, it makes it easy to change the percentage later on if we choose to. The compiled output of the following Sass code would be the exact same as the compiled output in the previous *Extend* section, and so we won't repeat that here:

```
/* Example of using variables and a built-in function */

$active-color: green;
$active-border-color: darken($active-color, 20%);
$inactive-color: yellow;
$inactive-border-color: darken($inactive-color, 20%);
$terminated-color: pink;
$terminated-border-color: darken($terminated-color, 20%);

%common-status-styles {
  width: 200px;
  height: 75px;
  padding: 10px;
  color: #333;
}

.active {
  @extend %common-status-styles;
  background-color: $active-color;
  border-color: $active-border-color;
}

.inactive {
  @extend %common-status-styles;
  background-color: $inactive-color;
  border-color: $inactive-border-color;
}

.terminated {
  @extend %common-status-styles;
  background-color: $terminated-color;
  border-color: $terminated-border-color;
}
```

Custom functions

Sass gives us a lot of power through using its ready-to-use, built-in functions, but sometimes, there's no substitute for custom functions—functions that do exactly what you want for the project at hand. The folks on the Sass team knew this and thus gave us a way to add custom functions to our Sass files.

To wrap up this Sass crash course, let's take a quick look at how we can create a custom function. Our function will compute the width as a percentage given two parameters, the number of columns for the target width, and the total number of columns we have.

In this short example, you'll notice that we do the following:

- Make use of variables
- Perform some simple math
- Use a built-in Sass function (that is, `percentage`)
- Introduce two new Sass commands: `@function`, and `@return`:

```
@function column-width-percentage($cols, $total-cols) {  
  @return percentage($cols/$total-cols);  
}  
  
.col-1 {  
  width: column-width-percentage(4, 12);  
}  
  
.col-5 {  
  width: column-width-percentage(5, 12);  
}
```

This compiles to the following CSS:

```
.col-1 {  
  width: 33.33333%;  
}  
  
.col-5 {  
  width: 41.66667%;  
}
```

It's my hope that you'll find a place in your web development for Sass. It may seem like it's all just overkill at the moment, but when you take some time to play with it, I'm confident that you'll discover clever ways to use Sass to help you better organize your CSS, and have it help trim the fat of code duplication.

Let's now shift gears and take a quick look at Bootstrap.

Bootstrap crash course

In this section, we're going to take a look at Bootstrap, in particular, its responsive grid and its components. We're going to cover just enough of Bootstrap's grid to give you a solid start on how to use it. We're also going to cover just five of Bootstrap's components to get you started. Bootstrap has a lot more than five components, and there are many ways you can customize each one of them. However, this is a crash course on Bootstrap and not a comprehensive manual—which is what would be needed to even attempt to cover Bootstrap in any level of detail. Bootstrap is a vast library with tons of options for its use, and thus it is far more important to just show you the basics and to show you where to go for further information on Bootstrap than it is to attempt to cover it exhaustively. The good news is that this crash course on Bootstrap is the fastest way to get you up and running with it.

The reasoning for taking this approach is the following:

- We won't be using all of the Bootstrap components for our example application
- Our example application will also be made from ng-bootstrap components, and Angular Material components (which we'll be covering in later chapters anyway: [Chapter 8, Working with NG Bootstrap](#), and [Chapter 9, Working with Angular Material](#))
- The most important part of Bootstrap for us will be Bootstrap's Grid—and we'll be covering the grid in more detail than the five components we'll be looking at

However, unlike the Sass crash course, we will see how to use Bootstrap in a practical way, since we'll be using it directly in our example application as we lay out the home page in this chapter.

What is Bootstrap?

Bootstrap is a CSS framework for building responsive websites, with an emphasis on being mobile-first. While there are other frontend presentation frameworks, Bootstrap is still the king of the hill in this arena—not only due to being the one with the most mind share, but it probably has the most *runtime*. What I mean by runtime is the number of times it's been used in websites, and thus it has been put through its paces more than the other CSS frameworks. Bootstrap's leading mind share (that is, popularity) is primarily due to three things:

- It was one of the very first frameworks of its kind (so the competition was virtually non-existent)

- It came with the backing of one of the top social websites in the world (that is, Twitter)
- It has been around since August 2011, and so is mature

Additionally, as we'll see in Chapter 8, *Working with NG Bootstrap*, the ng-bootstrap project is all about creating Angular widgets using Bootstrap 4, which says a lot about what the Angular community thinks of Bootstrap.

There is a reason that this third edition remains true to the relationship between Angular and Bootstrap, and that is because they are each leaders in their respective niches, as well as being symbiotically compatible and complimentary. In fact, these two frameworks are all you need to build the frontend part of powerful web applications—choosing just about anything you like for building out the backend, since all backend frameworks these days can produce and consume JSON, including mainframes still running COBOL programs. This is because JSON has become the most popular way to integrate systems through messaging.

Motivation

If you've ever tried to build a website that works well on different viewport sizes (that is, form factors/screen sizes) without the use of a framework to help you along, the motivation for Bootstrap is pretty easy to see—building something like that from scratch is both tedious and difficult. Mobile computing really escalated the need for something like Bootstrap to come along, and it was inevitable that it would. While the same can be said for just about any framework, in that you probably shouldn't spend time reinventing the wheel unless you have an extremely good set of reasons for doing so, it can be argued that (for the vast majority of websites, and even web applications) the frontend has become more important than the backend. It's been a fact for the last several years that the client side is the new black. I'm not suggesting that the backend is not important—nothing could be further from the truth, and *Integrating Backend Data Services*, is completely devoted to building out the backend. However, I am suggesting that when mobile computing came to be, we already had more than enough backend technologies and scores of frameworks to choose from, but were lacking in frontend frameworks.

The final comment I will add to conclude this motivation section is that killing two birds with one stone in the business world can give companies a competitive advantage (that is, speed to market) and/or financial advantage (that is, cost savings)—and so it isn't any different with software development. If you can build something once, in this case, a series of web pages, and use the same client-side code for both mobile and desktop instead of building two sets of everything (or even three sets, considering tablets as well), you should realize savings of both time and money. That is the promise—unfortunately, it is not always fulfilled. However, getting some advantage in these areas is certainly better than getting none at all.

Bootstrap's role in our example application

For our example application, Bootstrap will be used for only two purposes:

- To lay out the web page using its responsive grid
- To leverage, some of its read-to-use components for quickly building a nicely styled UI

Installing Bootstrap

Installing Bootstrap, for the purposes of learning it, will be different from how we'll installing-bootstrap in our Angular application. This chapter focuses on Bootstrap's grid system as well as some of its components, and so we're going to keep things simple for now by not creating an Angular application—or make any use of Angular at all, just yet. By the end of this chapter, we'll just have our skin and bones application (as previously mentioned), which we will then be converting into a fully-fledged Angular application.

Let's get started with the minimal, and fastest, way to integrate Bootstrap into our HTML. To use all Bootstrap has to offer, we only need to add resource links to one style sheet, and three JavaScript files.

The following is the HTML code that creates an empty HTML page that demonstrates the bare essentials to wire in Bootstrap:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chapter 3 - Bootstrap: Responsive Grid Layout &
          Components</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
          bootstrap/4.0.0/css/bootstrap.min.css" crossorigin="anonymous">
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js">
```

```
        crossorigin="anonymous">></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/
    1.12.9/umd/popper.min.js" crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
    js/bootstrap.min.js" crossorigin="anonymous"></script>
</head>
<body>
This page is intentionally blank. It's sole purpose is to show the HTML
code that needs to be added to integrate Bootstrap.
</body>
</html>
```

Following the order of the linked files in the previous HTML code, here are what the one CSS file and the three JavaScript files are for:

- The `bootstrap.min.css` file is the minified style sheet for Bootstrap, which is where all the default styles are defined
- The `jquery-3.2.1.slim.min.js` file is the minified JavaScript file containing the jQuery library and is referenced because Bootstrap itself relies on jQuery
- The `popper.min.js` file is the minified JavaScript file for yet another third-party library called Popper and is referenced because Bootstrap makes use of the functionality therein for its Tooltip components
- And, finally, the `bootstrap.min.js` file is the minified JavaScript file for Bootstrap itself and is used for various components, such as the modal and drop-down components, which require JavaScript to function

You'll also notice that the links are resources to CDNs (that is, content delivery networks). Although there are other ways of installing Bootstrap in our website, the advantages of using CDNs are threefold:

- We don't need to download and include the files in our web project
- The load time for the clients loading our web pages is minimized on account of the fact that these resources may have already been downloaded to their browsers while visiting other sites before ours
- The servers are optimized for delivering those assets (using caching and other hosting strategies)

We will look at adding our navigation bar on our home page when we take a look at the Navs and the Navbar components later in this chapter.

Bootstrap's responsive grid system

From my perspective, especially as a web developer that focuses on Angular, the most important thing that Bootstrap offers is a responsive grid. The reason for this is that there are many web / HTML components from many different libraries to choose from for web development (such as NG Bootstrap and Angular Material, both of which we'll be covering in later chapters) and, hence, we're not at all limited to just using Bootstrap's components. However, no matter which components you end up using, or if you create your own (as we'll learn how to do in [Chapter 6, Building Angular Components](#)), the Bootstrap grid can still be used for building responsive layouts and greatly simplifying our laborious task of creating well-designed web applications.

Bootstrap's grid makes it really easy for us to lay out our pages for use with various viewport (that is, screen) sizes. We simply use special Bootstrap classes to dictate how things get positioned on our pages for the different viewport sizes that our applications may be running on.

If you've ever wondered if Bootstrap is built on top of anything else, the answer is, unsurprisingly, *yes*. Libraries and frameworks often piggy-back on one another. This is how modern software in the open-source world is built. After all, why reinvent the wheel when there are perfectly sound wheels already available for our use? We have already seen (from the previous section on installing Bootstrap) that Bootstrap relies on jQuery and Popper. Bootstrap's responsive grid system is built on top of the CSS Flexbox, which was introduced in CSS3.

There is a newer grid system in CSS4, called the CSS Grid, but Bootstrap 4 does not use it. Bootstrap uses CSS Flexbox. This does not mean that Bootstrap is behind the times because newer does not necessarily mean better. Some things are made easier using the CSS Grid, and other things remain easier using CSS Flexbox.

A bit later on, when we discuss Bootstrap's predefined classes that are used for aligning things vertically and horizontally within the grid, the class names may seem familiar to you if you are familiar with CSS Flexbox. This is because Bootstrap uses CSS Flexbox under the covers, and the class names were inspired by its class names.

The grid itself has three main parts to it (container, row, and column), and each one of these is a class defined in Bootstrap's CSS file—which is why it needs to be referenced in our pages.

This is where the vernacular gets a little confusing, so let me explain something real quick. As you may know, there are no HTML elements named `container`, `row`, or `column`. We do, however, have the `div` element in HTML, and, in Bootstrap, we decorate it with a class—particularly, with a class of `container`, `row`, or `column`. But when we are talking about Bootstrap, it would be easier to pretend that there are HTML elements of these types. Let me clarify this because I'll be referring to the Bootstrap `row` as the `row` element, and the Bootstrap `column` as the `column` element from here on. Here's what I mean:

- It is easier to say *the container element*, rather than have to say *the div element with a class of container* (in code, this is what *the container element* looks like: `<div class="container">`)
- It is easier to say *the row element*, rather than have to say *the div element with a class of row* (in code, this is what *the row element* looks like: `<div class="row">`)
- It is easier to say *the column element*, rather than have to say *the div element with a class of column* (in code, this is what *the column element* looks like: `<div class="col">`)

OK, I hope that makes sense.

The container

The `container` is the root, or top-level element, in the grid. It holds one or more `rows`, which must be nested within the `container`, and the `rows`, in turn, can contain zero or more `columns`. To create a Bootstrap grid, we start off by creating a `container`—and to do that, we simply create a set of HTML `div` elements and assign the `container` class to the first `div` element.

Here is what this looks like in code:

```
<div class="container">  
</div>
```

Ha! Do you see why I mentioned the previous stuff about a `container` element? It's a messy way to try and explain it. So, let's rephrase that now using our new vernacular.

To create a Bootstrap grid, start off by adding a `container` element like this:

```
<div class="container">  
</div>
```

Ah, that is much easier to say and read! OK, back to our regular scheduled program...

There are two types of containers that you can have—and it is their class name that differentiates them from one another:

```
<!-- fixed-width container centered in the middle the viewport
--> <div class="container"></div>

<!-- full-width container that spans the entire viewport width (no margins)
--> <div class="container-fluid"></div>
```

The row

The row element must be nested within the container element. (Ha! I love this element stuff. Just try and explain it in writing without doing something like this!) A Bootstrap grid must contain at least one row and can contain as many rows as are needed.

Adding to the previous container code, here is what the code for a grid looks like that has two rows:

```
<div class="container">
  <div class="row">
    </div>
  <div class="row">
    </div>
  </div>
```

A row does not have to contain a column at all—for example, you may simply just want empty space between two rows in your grid—but it can have a maximum width of 12 columns within it.

However, it is important to note that the number of columns a row has is not proportional to the number of nested column elements (we'll look at Bootstrap's notion of a column in the next section). This is because the total number of columns in a row is independent of the number of column elements in that row.

Let me show you three examples to clarify this notion by adding to the previous row code. I will explain what `class="col-4"`, `class="col-6"`, and, generally speaking, `class="col-x"` (where x is an integer ranging from 1 to 12), all mean right after the following three grid examples.

In the first example, the grid has two rows, each of which has three columns of equal width:

```
<div class="container">
  <div class="row">
    <div class="col-4">
    </div>
    <div class="col-4">
    </div>
    <div class="col-4">
    </div>
  </div>
  <div class="row">
    <div class="col-4">
    </div>
    <div class="col-4">
    </div>
    <div class="col-4">
    </div>
  </div>
</div>
```

In the second example, the grid only has one row with two columns of equal width:

```
<div class="container">
  <div class="row">
    <div class="col-6">
    </div>
    <div class="col-6">
    </div>
  </div>
</div>
```

In the third example, the grid also only has one row with two columns, but they are not of equal width. In fact, the first column only takes up 25% of the row's total available width, and the second column takes up the remaining 75%:

```
<div class="container">
  <div class="row">
    <div class="col-3">
    </div>
    <div class="col-9">
    </div>
  </div>
</div>
```

OK, now that we've seen three grid examples, we can discuss what on earth the "col-x" class names mean. The grid allows up to 12 columns per row, and each column element you embed in the row can span 1 to 12 columns—and is what the x represents. As a quick example, if we have a column element in our row, and we'd like for it to span eight of the available 12 columns, our class name would be col-8 and our column element would look like this: <div class="col-8">. The key is that the total number of columns in our row (which is the addition of the x's in our class names) should not surpass 12. However, it can be less than 12.

Additionally, each row in your grid can have different numbers of columns, where each column is also a different width to the other columns within the same row. Let's quickly look at an example before discussing some interesting ways that you can align columns within the row by adding pre-defined Bootstrap classes to the row element:

```
<div class="container">
  <div class="row">
    <div class="col-10">
    </div>
    <div class="col-2">
    </div>
  </div>
  <div class="row">
    <div class="col-4">
    </div>
    <div class="col-3">
    </div>
    <div class="col-5">
    </div>
  </div>
</div>
```

The grid in the preceding code has two rows, where the first row has two columns of unequal widths, and the second row has three columns of unequal widths.

Whenever you have a grid, you need to concern yourself with how things align up within it. Bootstrap has pre-defined classes that can be used on the row elements in order to align the column elements within them.

Here are a few of these classes:

- justify-content-center (centers the columns)
- justify-content-start (left-justifies the columns)
- justify-content-end (right-justifies the columns)
- justify-around-end (spaces the columns evenly)
- justify-between-end (puts all the available space between the two columns)

The interesting thing about these classes—which affect the horizontal alignment of the columns in the encapsulating row—is that you can only see their effect when the total spanned columns of all the column elements amount to a number less than 12. This is precisely why having fewer than 12 spanned columns is allowed.

Here is an example of what a row element that contains fewer than 12 spanned rows looks like:

```
<div class="container">
  <div class="row justify-around-end">
    <div class="col-4">
    </div>
    <div class="col-4">
    </div>
  </div>
</div>
```

In this previous example, we have a grid that has one row and that row contains two columns. However, since the total spanned columns are fewer than 12, the horizontal alignment that will be applied (due to the `justify-around-end` class) will have a visible effect—which is to center the columns while inserting the available unused space (a third of the row's width in this case) around the columns. This will have the appearance of margins on either side of the columns, with double the margin between them.

The other classes mentioned have different horizontal alignment effects than the description beside them in the bullet list. I encourage you to play around with these classes to get used to them.

The column

The column elements must be nested within the row elements, as the previous examples showed. We've already seen how many column elements can fit in a row, which depends on their respective column widths.

The columns in the grid are basically your cells in the grid and are where your content (that is, text, images, and so on) is to be inserted. If you have a grid that has six rows of four-column elements each, you have 24 cells in which to place your content.

Just as you can align the column elements within the row elements, using special classes on the row elements, you can also align the content within your column elements, using special classes on the column elements.

Here are some of the classes you can use on the column elements in order to align your content within them:

- `align-self-start` will force that specific cell's content to the top of the cell
- `align-self-end` will force that specific cell's content to the bottom of the cell
- `align-self-center` will force that specific cell's content to the vertical center of the cell

Differing viewport sizes

The final thing that I would like to cover regarding Bootstrap's grid is probably the most important one of all. What makes the grid responsive? That is to say, how does the grid adapt to different viewport sizes? The answer to this is two-fold. Firstly, most HTML layouts (even vanilla layouts that were not at all designed to be responsive) have some built-in leeway on how they react when viewed on screens of varying sizes. However, while the layout of a standard web page may still be acceptable in the way a browser renders it on a tablet versus a regular 19-inch monitor, things tend to break down and not work at all for a website that looks decent on a tablet, but is currently being viewed on a standard mobile phone, such as the iPhone 7, or a similarly sized Android device. This is where we need some design intervention, and what leads to the second way in which a Bootstrap grid adapts to the device's viewport size—namely, special tweaks to the classes, and class names for the column.

You'll remember that the class name we've been using for the column element has the following general form:

```
<div class="col-x">
```

Well, in order to make the grid responsive, Bootstrap has included the ability for us to tweak the classes by adding a symbol to the class name between the `col` and the `x` (that is, an integer from 1 to 12).

For example, here is what the column element's class would look like with one of these symbols (in actuality, it's not a symbol but rather a new class name—however, for the purposes of explaining it, you can think of it as a symbol):

```
<div class="col-sm-4">
```

I'll explain what the `sm` in `col-sm-4` means in a moment, but, in practice, you'll see more than one class name on a column element. For instance, here is a likely set of class names on a column element:

```
<div class="col-xs-12 col-sm-4 col-md-3" >
```

OK, let's decipher what this set of classes is for. To do that, let me first list the available symbols and what they mean:

| Viewport size | Extra-small | Small | Medium | Large | Extra large |
|---------------------|-----------------------------|---------------------------|---|---|--|
| Grid breakpoint | <576px | >=576px | >=768px | >=992px | >=1200px |
| Max container width | None | 540px | 720px | 960px | 1140px |
| Symbol | xs | sm | md | lg | xl |
| Typical devices | iPhone, iPod, Android phone | iPad 1, iPad 2, iPad Mini | Older monitor (low res, 800x600), a few older Android tablets | Regular modern monitor, most modern Android tablets | Hi-res modern monitor, iPad 3, iPad 4, Android tablets |
| Class prefix | .col-xs- | .col-xs- | .col-md- | .col-lg- | .col-xl- |

In the preceding table, in the third row from the bottom, I've listed the five symbols that are available to you. In the second row from the bottom, I have listed the typical target devices for which the symbol, and thus the grid-breakpoint, applies. I'll discuss the grid-breakpoint in a moment, but I just wanted to say that these target devices I've listed are rules of thumb—they are not set in stone. For instance, Android tablets are listed in three of the five viewport size columns. This is because there are many Android tablet manufacturers and even more sizes of displays (that is, viewports) that they come in. The same can be said for laptops. However, the viewport sizes on Apple-based products are well known, and fewer in number—and is why I listed them by name. Suffice it to say, by taking a look at the typical device's row, you can get a fairly good idea of what column class you probably want to use.

Armed with this knowledge of viewport sizes and the previous table, let's now decipher what this column element and set of classes mean:

```
<div class="col-xs-12 col-sm-4 col-md-3" >
```

This column element contains a set of three classes, and each class basically instructs the browser on how to render the column and its contents depending on the viewport size. As an aside, technically speaking, the viewport size is the maximum dimensions (in pixels) of the display. Taking the case of a 13-inch laptop monitor with its resolution set to 1600 x 900, its viewport size is 1600px wide by 900px high. Practically, however, the viewport size is the dimensions of the browser window, and not the laptop's display itself. This is an important distinction when we talk about responsive web design because, when using a desktop or laptop computer, people can resize their browsers—which forces the web page to be re-rendered—and, hence, this is truly what the viewport size is, from Bootstrap's perspective, and for our purposes.

Coming back to the deciphering of the previous column element, while referencing the previous viewport size table, and having mentioned how the resizing of the browser dictates the viewport size that we, as developers, care about, we can now decipher what these three classes are instructing the browser to do:

- `col-xs-12`: This tells the browser that when the viewport is fewer than 576 pixels in width, the column should span all 12 columns. In other words, the column should consume the entire available width of the row.
- `col-sm-4`: This tells the browser that when the viewport is between 576 and 767 pixels in width, the column should span four of the 12 available columns. In other words, the column should consume 1/3 of the row's width.
- `col-md-3`: This tells the browser that when the viewport is 768 or more pixels in width, the column should span three of the 12 available columns. In other words, the column should consume 1/4th of the row's width.

We could have controlled the rendering of the column for viewport sizes 992 or more pixels in width by adding the classes with the class prefixes `.col-lg-` and `.col-xl-`, but, in the example we've just seen, we didn't seem to care—which is to say, no matter how wide the viewport is (even 2400px!), our column's width would scale to consume 25% of the row's width.

And that, ladies and gentlemen, is how you can design a web page while maintaining how the contents in your grid's cells are to be rendered on thousands of viewport sizes. By leveraging Bootstrap's grid, we no longer need to code up several variations of our web pages to get them to display the way we want on different sized displays. Pretty darn cool, ain't it?

Bootstrap components

As previously mentioned near the beginning of this chapter, I did not want to simply regurgitate Bootstrap's documentation when covering components. Instead, I will briefly discuss five of Bootstrap's components that we'll be using, showing some basic code for them, and pointing you to Bootstrap's official documentation for these components so you can learn more about their options, of which there are many—far too numerous to cover in a book such as this fine example you are currently reading.

Button components

Buttons are all around us—and no, I'm not referring to the buttons on your favorite dress shirt. If you've ever been on an elevator (hey, some people absolutely refuse to step into one), you will no doubt see buttons—and pressing one of them will transport you to a place you want to go. The same thing goes with your TV remote—but instead of transporting you (at least not quite yet—but maybe in the future, y'all just never know), it transports your mind to another place from the very comfort of your living room. These buttons perform functional, meaningful tasks. How about buttons on a web page? Well, arguably, they also transport things—such as information, when you click the submit button on a form that you fill out. But perhaps an equally important feature of buttons is to help make your web pages attractive, and intuitive. Luckily, Bootstrap makes it super easy to add beautifully styled buttons to our web pages—100 times more refined than the default gray button that a browser renders when you add a button element.

Let's take a look at a few of these while exploring some of Bootstrap's pre-defined button classes (that is, styles).

Out of the box, without any tweaking necessary, we can easily insert a beautifully styled button by assigning two classes to the button element like this:

```
<button type="button" class="btn btn-primary">Click me</button>
```

That button is blue in color, but there are other default colors we have access to via these other classes:

- **btn-secondary**: Light charcoal gray, with white font
- **btn-success**: Light green, with white font
- **btn-danger**: Red, with white font
- **btn-warning**: Goldenrod, with black font

- `btn-info`: Teal, with white font
- `btn-light`: Light gray, with black font
- `btn-dark`: Almost black, with white font

There's also a class for turning a button into a link: `btn-link`

If you prefer something more white, or less heavy on the color, Bootstrap has a set of classes that match the preceding classes called *Outline buttons*. The colors and class names are the same, with the only difference being the word *outline* between *btn* and *secondary*, *success*, *danger*, and so on. The buttons are transparent except for the outline, or border, and of course, the font color for the text on the button.

Here's what these class names look like:

- `btn-outline-secondary`: Light charcoal gray outline, with the same color for the font
- `btn-outline-success`: Light green outline, with the same color for the font
- `btn-outline-danger`: Red outline, with the same color for the font
- `btn-outline-warning`: Goldenrod outline, with the same color for the font
- `btn-outline-info`: Teal outline, with the same color for the font
- `btn-outline-light`: Light gray outline, with the same color for the font
- `btn-outline-dark`: Almost black outline, with the same color for the font

All these buttons come in a default size in terms of height, and font size. However, as you may have guessed, Bootstrap has a way in which to make the default button larger or smaller by adding the `.btn-lg` or `.btn-sm` class, respectively. Here is what that would look like:

- `<button type="button" class="btn btn-primary btn-lg">I'm large</button>`
- `<button type="button" class="btn btn-primary btn-sm">I'm small</button>`

You can read all you care to about Bootstrap's buttons here: <https://getbootstrap.com/docs/4.0/components/buttons/>

Alert components

When a user takes an action on a web page, such as updating their phone number on their user profile, it's always nice to let them know if the update was successful or not.

Sometimes these user feedback messages are referred to as "flash messages" (because they often only appear for a few moments, and then fade away so as to not clutter the screen). Bootstrap calls them "alerts", and they are created by adding the predefined alert classes and a role attribute to a `div` element.

For the most part, their coloring and naming scheme is fairly consistent with the button components. Here are the alerts that are available:

- `<div class="alert alert-primary" role="alert">This is a primary alert</div>`
- `<div class="alert alert-secondary" role="alert">This is a secondary alert</div>`
- `<div class="alert alert-success" role="alert">This is a success alert</div>`
- `<div class="alert alert-danger" role="alert">This is a danger alert</div>`
- `<div class="alert alert-warning" role="alert">This is a warning alert</div>`
- `<div class="alert alert-info" role="alert">This is a info alert</div>`
- `<div class="alert alert-light" role="alert">This is a light alert</div>`
- `<div class="alert alert-dark" role="alert">This is a dark alert</div>`

Not only do Bootstrap's alerts look pretty, but they are quite neat. You can embed links in them (since it's just HTML, after all), and even insert an optional dismiss button. The alert component is a good example of why Bootstrap depends on the jQuery library, since it's required for the dismissal of the alert component.

Alerts are worth the time to learn so that you can utilize them in your applications. Here is the link to Bootstrap's documentation on its alert components: <https://getbootstrap.com/docs/4.0/components/alerts/>.

Navbar component

The Navbar component is very rich — you can do a lot with it—but, in essence, it's Bootstrap's way to give you a nicely styled navigation bar across the top of your web page. The richness comes from the fact that there are a few sub-components that can be used. These include the following:

- .navbar-brand for your company, product, or project name
- .navbar-nav for a full-height and lightweight navigation (including support for dropdowns)
- .navbar-toggler for use with our collapse plugin and other navigation toggling behaviors
- .form-inline for any form controls and actions
- .navbar-text for adding vertically centered strings of text
- .collapse.navbar-collapse for grouping and hiding navbar contents by a parent breakpoint

Showing examples of all these items here would be too costly, with little associated benefit. Rather than do that here, I will show you how to use Bootstrap to build the navigation menu for our example application later in this chapter. The code can be found in the code listings at the end of the chapter. The first wireframe in the pages that follow shows a logo placeholder, a menu, and **Login** and **Try Now** buttons. The wireframes represent drafts of the pages we're aiming to build. Our navigation bar will look slightly different, but will encompass all the parts that are shown on the wireframe.

More documentation on Bootstrap's Navs and Navbar components can be found: <https://getbootstrap.com/docs/4.0/components/navs/> and <https://getbootstrap.com/docs/4.0/components/navbar/>.

Modal components

Modal components are great ways to draw your user's attention to things by using them for creating lightboxes, user notifications, and more. I like to use them for popping up forms for users to add items and edit them as well, directly from the page that lists those items. This way, all the functionality (that is, view, add, edit, and delete) for a listing of items is all done on one page. Using the modal component in this way leads to a clean design that is intuitive to users.

As with the Navbar component, showing examples here is not the best way to demonstrate modals. Rather than show contrived examples, I will show you through code (by referencing the code listings at the appropriate time) how we'll achieve creating the modal forms shown in the following wireframes. When you take a look at the wireframes, you'll see that I've used modals quite generously throughout the pages. I've even used them for the login and the sign-up functionality of the site.

There are several demos on Bootstrap's modal component that you can check out here: <https://getbootstrap.com/docs/4.0/components/modal/>

We've only covered four commonly used components that Bootstrap offers, but they are enough for us to get a glimpse of what can be done with pre-defined components. There are many other components that can be used, and you can find them on the official Bootstrap website here: <https://getbootstrap.com/docs/4.0/components/>

Again, we're not covering all of Bootstrap's components because the official documentation has done that job already — and has done it well. Additionally, we'll be using NG Bootstrap components, Angular Material components, and custom components that we'll be creating together in later chapters.

Listing Carousel – a formal introduction

The evolution of a software project is a very interesting thing, and it follows a very logical series of stages. Here are the stages that we will cover—which are true of any software project:

1. Idea generation/concept.
2. Analysis/feasibility study: The purpose of conducting a feasibility study on the product concept is to vet the ROI (that is, return on investment) for the project. In other words, is the project worth the company's investment of its resources (time, money, talent, and so on)?
3. Requirement gathering.
4. Use cases.
5. Wireframes.
6. Implementation.

With these software project stages outlined, let's look at a real-life example using Listing Carousel.

Idea generation/concept

Ideas for software projects can come from anywhere and at any time, but for the vast majority of the time, the ideas are inspired by a need to solve problems that organizations will invariably have from time to time throughout their lifetimes. The two main categories of problems are addressing an inefficiency and creating an opportunity for itself in the market via creating a competing product that is better than (that is, differentiated from) those of its competitors. Stated in other terms, software projects can often be viewed as either being an efficient play or a competitive advantage play. Solving these two types of problems are the needs that every growing organization will have at some point, or at many points, in time throughout its existence.

So, how was Listing Carousel conceived? Having been a real estate salesperson in a previous life, turned IT pro, it was easy for me to think of a way to develop a better way for real estate agents to spread the word of their new listings (that is, properties for sale) to their social media circles, and to showcase their listings in a more informative way than the other primary options currently available to them. While there are several ways a real estate agent can market their listings, I found they lacked two basic things:

- A way to easily spread the word of their listings to their social media circles (that is, Instagram and Facebook)
- A way to present the properties in a slightly more engaging way, while also better describing the properties at the same time

So, the problem I had was that I had to create a software product that was markedly differentiated from other software services. The solution was to think of the two product differentiators listed previously and to assume that I can get access to the technology required to make it happen. Thus, for Listing Carousel, you can say that the software project was conceived as a competitive advantage play.

Great! I had a potential software project to work on! What's next? Well, as mentioned at the start of this section, the next stage was to do a feasibility study. Recall that the purpose of conducting a feasibility study on the product concept is to vet the ROI for the project, and to also do the research to see whether the required technology was readily available—or, if not, can it be created? Let's briefly look at this next.

Analysis — feasibility study

This stage in the project analysis is where the go / no-go decision will be made. The proper way to conduct a feasibility study is to prepare a business plan, and present it to investors. Why? The reason why managers of a company go through the process of writing a business plan, and showing it to investors (or to the VP, President, or CEO of the company—for internal software projects), is because they need a document that they can share with an investor to gauge the interest in the project. If the investors would be interested in making an investment, then it means that the project has merit.

The perfect document for this is a formal business plan because it contains summary information on all the important things investors would want to see, namely:

- **Market analysis:** Is there room in the market for yet another similar product? What is the market potential?
- **Competitive analysis:** How is our product/service going to be different? Are we going to compete on cost, quality, or features?
- **Required resources:** What personnel does the project need? And how many man-hours to build it and deliver it to market?
- **Budgets:** How much money in total should be budgeted for the project (IT, sales, operating costs, and so on)?
- **Financial projections:** What revenues can be expected over the next 12 months, two years, three years, and five years? What is the break-even point?
- **Exit strategies:** How long do we operate the company for? How do we get our investment out?

You may be asking yourself if I actually prepared a detailed business plan for a software project that was fairly small in size. The answer—of course! Why? In short, I needed to see whether implementing the project was worth my time and money. Specifically, I spent the necessary time preparing a business plan for the following reasons:

- **Market analysis:** As good as an idea may sound to you, you need to do your due diligence in being reasonably sure that there is a need for yet another product or service in the market you are entering. If there is room, then you have a potential opportunity. In my case, I believed that room existed for Listing Carousel and that it was sufficiently differentiated to give me a competitive advantage over the competing products in the market.

- **Cost and time to develop:** Time and money are valuable commodities—and developing a software product or service will take both. Every dollar you invest in one project means that you can't invest it in another. And the same is true of your time. Every hour you spend on doing something means you gave that time up for doing something else instead. So, choose where you place your resources wisely! In my case, I had some money earmarked for a fun project to do—so the money part was taken care of. How about the time? This was a difficult decision for me. While I did not really have the time, I liked the project and I have friends that are real-estate agents—and so I decided, what the hell, let's go for it. So, I knew how much money I needed to invest, and roughly how much time I had to invest as well.
- **Projected revenues:** Just because the required resources (that is, time and money) I had to invest were acceptable to me, it wasn't a done deal yet. The next step was to make some calculations to see whether I would make a profit over time, and how much. If the ROI was high enough, it was a go-ahead. In my case, the ROI was actually not as good as I wanted it to be—in fact, it was almost zero! In other words, I would just break even if I was lucky. However, you also have to listen to your gut, and my gut was telling me that I may be able to sell the software/service at some point, which would make the project worthwhile. At the time of writing, I have not yet sold Listing Carousel, but it did start to make a little bit of a profit.
- **Exit strategy:** Before you embark on building any business—and I treated Listing Carousel as a standalone business—you have to think of an exit strategy. What is an exit strategy? It basically defines how you will divorce yourself of your obligations of operating and/or servicing the company. Companies don't run themselves, and so unless you want to stay married to the company forever, you need to have an exit strategy from the start. I can't take more space in this book to outline this in detail, but suffice it to say, I structured the company in such a way where my exit strategy was baked in.

Requirement gathering

This stage of the software project forms the basis of your project plan, which is what a project manager uses to keep the project on schedule and on budget. Requirements are typically gathered from the end client (internal or external), but can also come from an idea board if you are building something new that does not yet exist in the market.

For instance, for Listing Carousel, I picked the brains of a few real-estate agent friends, telling them what I wanted to build, and how I wanted to make it different from what they were already using. Here is a partial list of what we came up with as requirements:

- The ability to create a carousel-style photo viewer (one per listing/property, with any number of photos in it)
- The ability for the user to upload photos
- The ability to annotate each photo (that is, a caption at the bottom of the photo)
- The ability to flip the photo around to show a detailed description of what is shown on the photo
- The ability for the user to link a photo to a carousel
- The ability for the user to order/re-order the photos within a carousel
- The ability for the user to post the carousel of the listing on Facebook
- The ability for the user to post the carousel of the listing on Instagram
- The ability to have a magic link manually placed on any website that the user has access to, which opens the listing's carousel in a modal lighthouse in place
- The ability to have a soundtrack play while the listing's carousel auto-scrolls through the photos in it
- Each carousel will have a unique short permalink assigned to it so that the user can email or text it to whomever they like

We won't build all these features into our example application because there just isn't the space in the book to do so—but we will build the important ones. The two features that we'll omit are the magic link and the playing of the music soundtrack. I won't make any promises, but I may create a blog post on how to build the magic link on my blog, *AngularMotion.io*, in the near future.

Wireframes

This stage of the process is where the look and feel of the application are planned and laid out.

The following are the 12 wireframes for the pages we'll be building (note: a couple of the wireframes were too long to have as one screenshot, such as for the Welcome page, and so there is more than one screenshot for them).

We will be implementing some of these wireframes in the following sections, and we will learn to implement some of the layouts and components in chapters to follow.

Implementation

This is where the rubber hits the road. We're going to code up some of our web pages using the wireframes we just reviewed to help guide us. We're also going to need a web server so we can serve up our new pages in our browser as we build them out.

Installing our interim web server

We'll be using Node's built-in web server for our project from Chapter 12, *Integrating Backend Data Services*, onward. But, since we've got a little while to go before we get to that point, we need to have a simple interim solution.

We've not spoken about browsers before because there wasn't a need to do so—but now, there's a need. While it doesn't matter which browser you use for viewing Angular applications, it would be easier—though not essential — for us to use the same browser while we work through this book together. My browser of choice, while I develop web applications, is Chrome. As with most browsers, Chrome has a ton of extensions that other developers have created that do everything from providing subscription notifications to debugging tools and more. You can download Chrome for the operating system of your choice from here: <https://www.google.com/chrome/>. You can search for and install extensions for Chrome from the Chrome Web Store at <https://chrome.google.com/webstore/category/extensions>. We'll be using Chrome, and specifically, a few of its extensions, for a few things in this book.

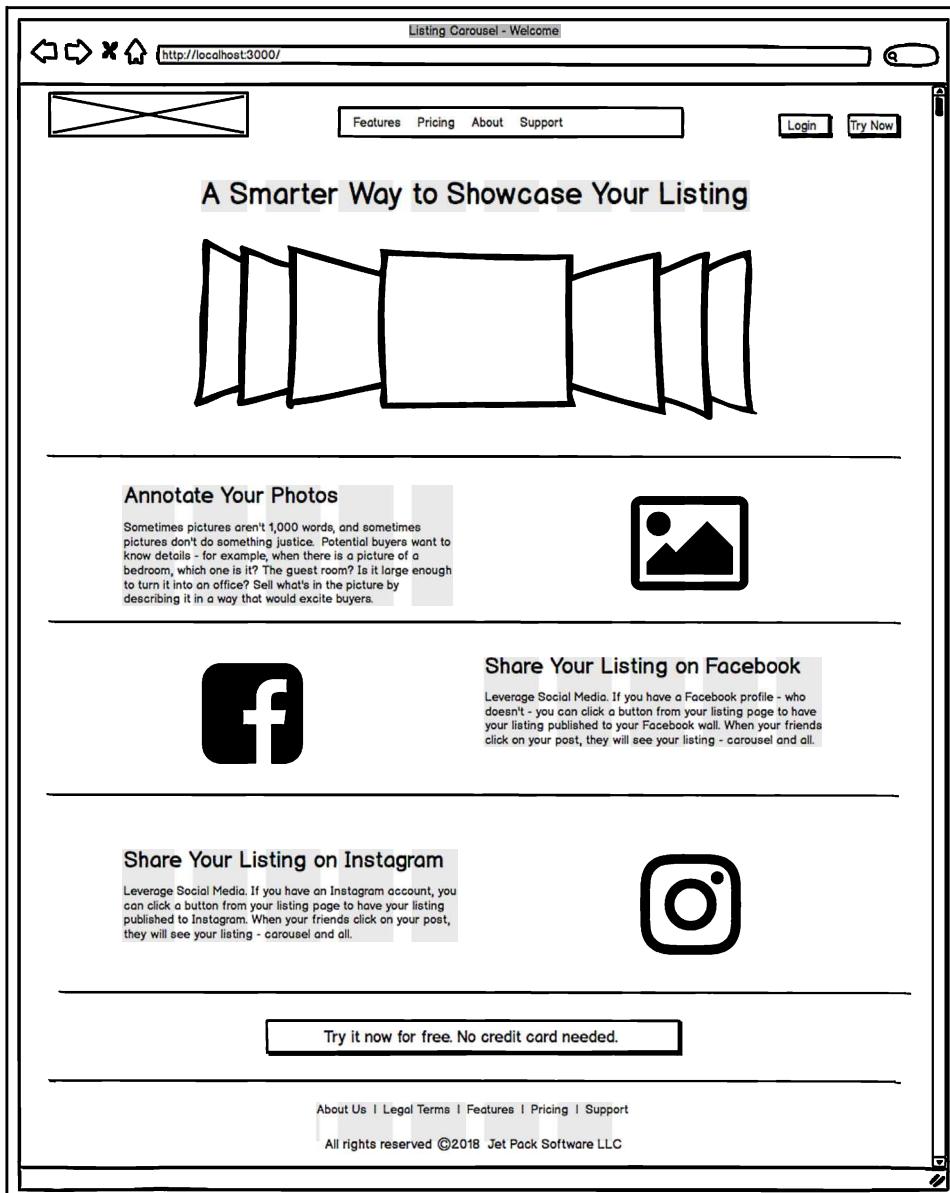
The first order of business is to install a Chrome extension that will help us serve up the pages we build for our application. It's called, *Web Server for Chrome* and you can search for it and install it from the Chrome Web Store. I didn't include the link to it directly because the URL was incredibly long.

This extension allows you to select a folder for where the files you wish to be served are located, as well as a port to listen on. There are other options that you can select as well. One common case option that is enabled by default is to have the extension automatically show the `index.html` file. For example, assuming you enter 8887 for the port number, you would point your browser to `http://127.0.0.1:8887` and would see that your `index.html` page in the folder you specify would automatically be served up in your browser. Once you have these two settings configured, you're off to the races and can view the pages we create.

Welcome page

The first wireframe we will implement using the Bootstrap components and grid layout is the Welcome home page.

Take a look at the following wireframe screenshot. We have a header section that contains our application logo placeholder, navigation menu, and **Login** and **Try Now** buttons on the right. This is followed by a jumbotron header that showcases the title of the application. Then, our content sections are divided so that we can add our content to the page:

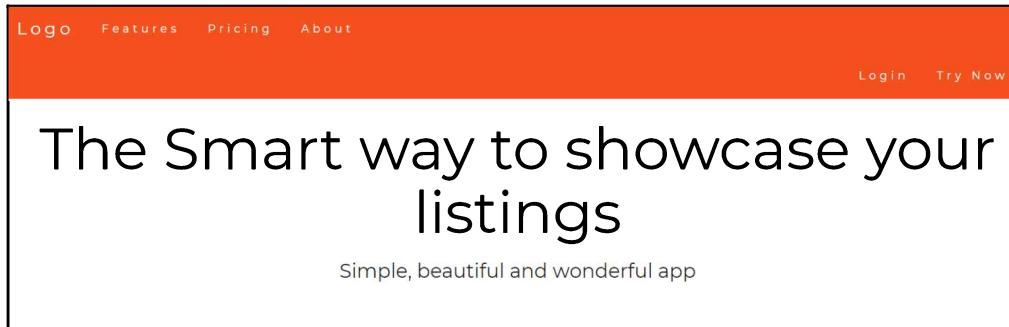


We will now proceed to implement our code implementation of the preceding wireframe screenshot. We are first going to implement the header section, and we will use the `<nav>` tag to categorize all of our header section code, including the logo, menu, and action buttons:

```
<nav class="navbar navbar-default navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
        data-toggle="collapse" data-target="#myNavbar">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#myPage">Logo</a>
    </div>
    <div class="collapse navbar-collapse" id="myNavbar">
      <ul class="nav navbar-nav mr-auto">
        <li><a href="#features">Features</a></li>
        <li><a href="#pricing">Pricing</a></li>
        <li><a href="#about">About</a></li>
      </ul>
    </div>
    <div class="collapse navbar-collapse ">
      <ul class="nav navbar-nav navbar-right">
        <li><a href="#features">Login</a></li>
        <li><a href="#pricing">Try Now</a></li>
      </ul>
    </div>
  </div>
</nav>
```

In the preceding code, we are implementing a `nav` tag element and using the Bootstrap `navbar` classes, `navbar-default`, `navbar-fixed-top`, `navbar-brand`, `navbar-collapse`, and so on. These classes come with default functionality that covers almost all aspects of a navigation section. One of the interesting things to note in the preceding code is the `navbar-collapse` class, which helps in rendering various device screen resolutions automatically. We have also added a few menu links for features, pricing, and about. We have also added our action items, **Login** and **Try Now**.

Launch the page in the browser and we should see the output, as shown in the following screenshot:



Next, we have to make our layouts for the content sections. The jumbotron and content sections. We will use the `jumbotron` class with a `div` section and, for the content sections, the Bootstrap grid column classes, `row`, `col-sm-8`, and `col-sm-4`:

```
<div class="jumbotron text-center">
  <h1>The Smart way to showcase your listings</h1>
  <p>Simple, beautiful and wonderful app</p>
</div>

<!-- Container (About Section) -->
<div id="about" class="container-fluid">
  <div class="row">
    <div class="col-sm-8">
      <h2>Annotate your prices</h2><br>
      <h4>Some pictures aren't 1000 words and sometimes pictures
          don't do something justice</h4><br>
    </div>

    <div class="col-sm-4">
      <span class="glyphicon glyphicon-signal logo"></span>
    </div>
  </div>
</div>

<div class="container-fluid bg-grey">
  <div class="row">
    <div class="col-sm-4">
      <span class="glyphicon glyphicon-globe logo slideanim"></span>
    </div>
    <div class="col-sm-8">
      <h2>Our Values</h2><br>
      <h4><strong>MISSION:</strong> Our mission lorem ipsum dolor sit amet,</h4>
    </div>
  </div>
</div>
```

```
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore  
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud  
exercitation ullamco laboris nisi ut aliquip ex ea commodo  
consequat.</h4>  
  
<br> <p><strong>VISION:</strong> Our vision Lorem ipsum dolor sit amet,  
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore  
et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud  
exercitation ullamco laboris nisi ut aliquip ex ea commodo  
consequat.</p>  
  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod  
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim  
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea  
commodo consequat.</p>  
  
</div>  
</div>  
</div>
```

Now, let's analyze the preceding code to understand some of the important points. We are making use of the awesome Bootstrap grid utilities to create our application layout, using the column classes to create the layout, which will render on various screen resolutions. Run the application in the browser and we should see the output as shown in the following screenshot:

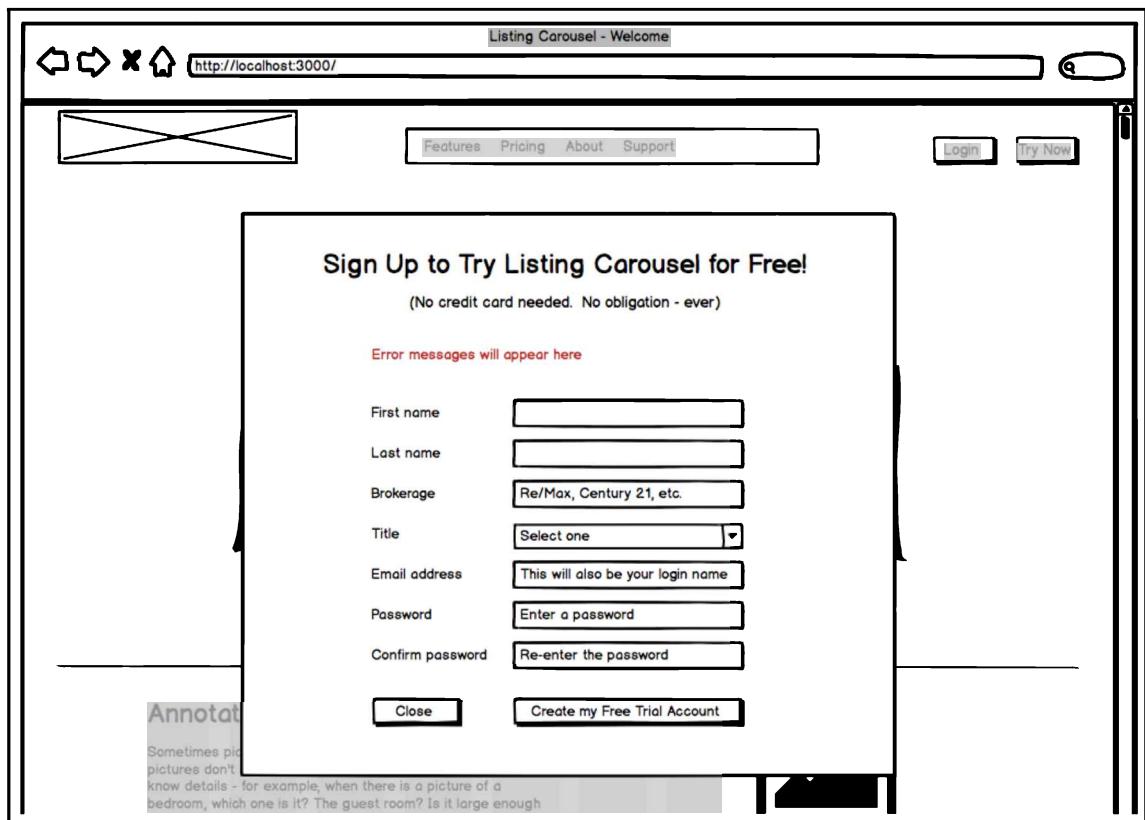
The screenshot shows a web application interface. At the top, there is a navigation bar with links for 'Logo', 'Features', 'Pricing', 'About', 'Login', and 'Try Now'. The main content area has a white background with a thin black border. It features a large heading 'The Smart way to showcase your listings' with a subtitle 'Simple, beautiful and wonderful app'. Below this, there are two sections: 'ANNOTATE YOUR PRICES' with a subtext 'Some pictures aren't 1000 words and sometimes pictures don't do something justice' and 'OUR VALUES' with a subtext 'MISSION: Our mission lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.' To the right of the 'ANNOTATE YOUR PRICES' section is a graphic of four orange bars of increasing height.

Nice work, fellas, so far. We have just created our first Welcome page layout using the Bootstrap layout components. We will continue to use the same and build some more wireframes to make you comfortable. In the next section, we will learn to create signup and login screens using the Bootstrap modal component.

Signup

Next up, we will implement our signup and login pages using Bootstrap's awesome modal component.

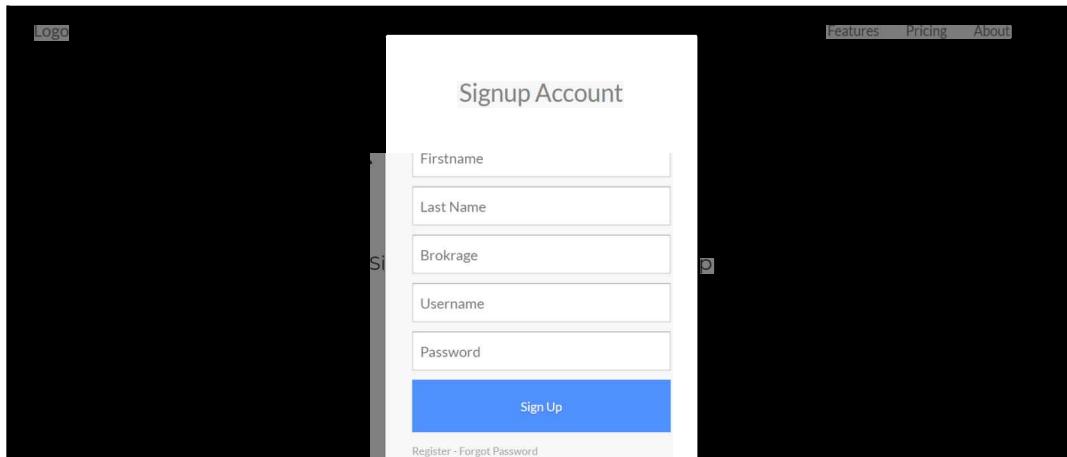
Take a look at the following wireframe. It's a simple modal window with some form field input elements:



Let's go ahead and implement the code. The following is the sample code for creating the modal window:

```
<div class="modal fade" id="signup-modal" tabindex="-1" role="dialog"  
aria-labelledby="myModalLabel" aria-hidden="true" style="display:  
none;">  
    <div class="modal-dialog">  
        <div class="loginmodal-container">  
            <h1>Signup Account</h1><br>  
            <form>  
                <input type="text" name="firstname" placeholder="Firstname">  
                <input type="text" name="lastname" placeholder="Last Name">  
                <input type="text" name="brokrage" placeholder="Brokrage">  
                <input type="text" name="user" placeholder="Username">  
                <input type="password" name="pass" placeholder="Password">  
                <input type="submit" name="login" class="login  
                    loginmodal-submit" value="Sign Up">  
            </form>  
            <div class="login-help">  
                <a href="#">Register</a> - <a href="#">Forgot Password</a>  
            </div>  
        </div>  
    </div>  
</div>
```

In the preceding code, we have used Bootstrap's modal component and modal classes modal and modal-dialog. Inside the modal dialog content, we have created our signup form with the input form elements—first name, last name, brokerage, user, and pass. Launch the page in the browser and we should see the output, as shown in the following screenshot:

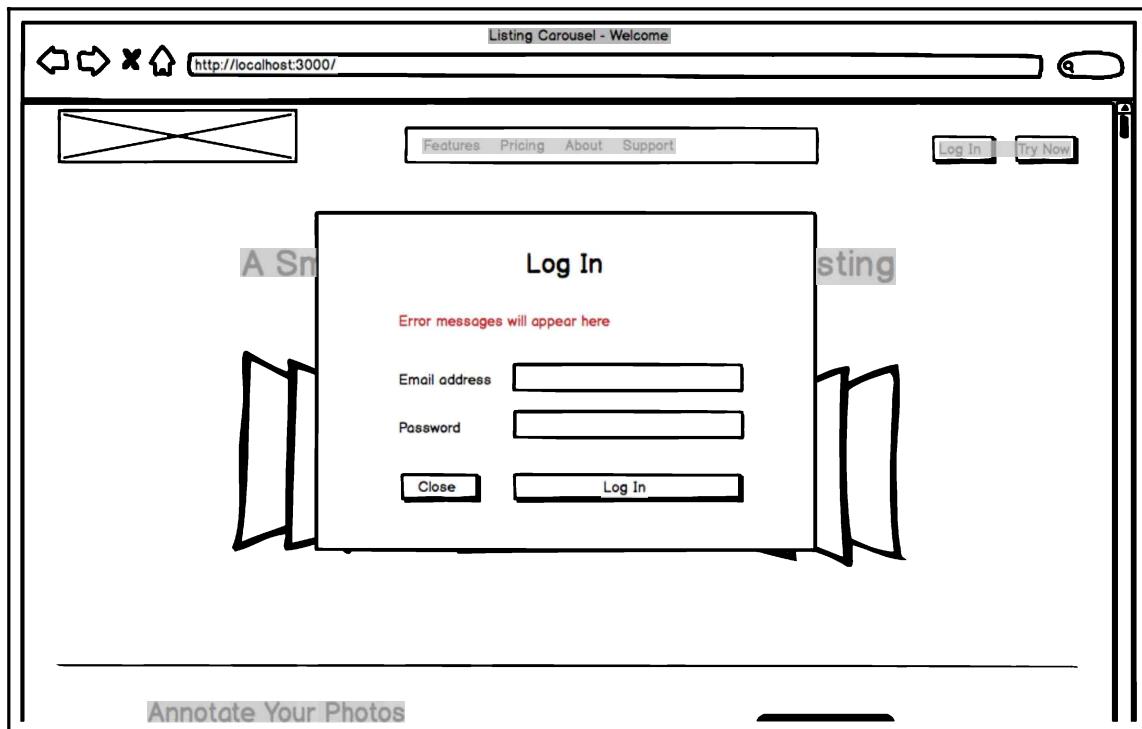


That's a great start to building our application. In the next section, we will build our login page using the same Bootstrap modal component.

Login

In the previous section, we have learned about creating the signup form inside a modal window. In this section, we will learn about creating a login screen inside the modal window. The methodology and principle are exactly the same as how we created the signup page.

Take a look at the following login wireframe, which we are going to implement in just a bit:



Time for some action. We are going to create a modal window first and we can bind a click event to open the dialog window:

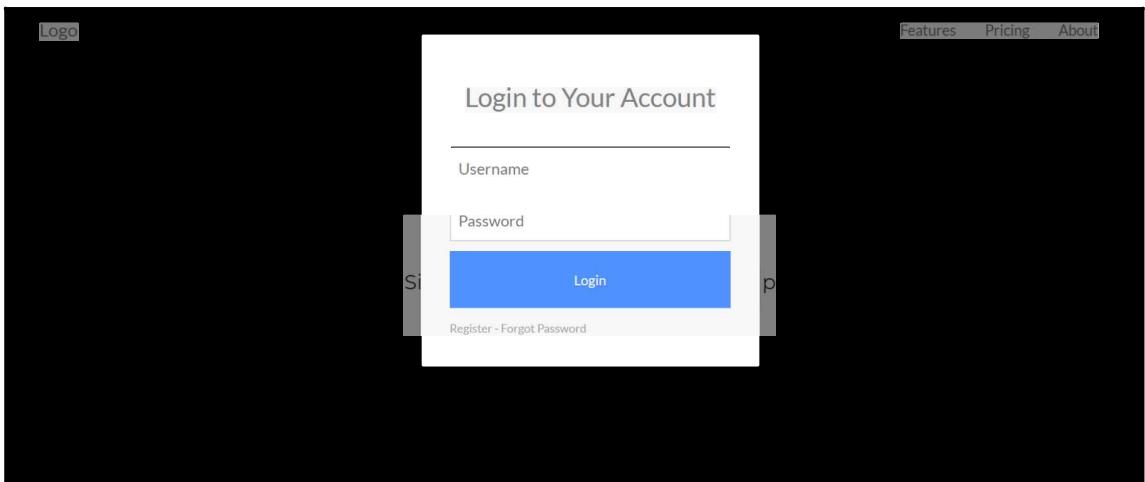
```
<div class="modal fade" id="login-modal" tabindex="-1" role="dialog"  
aria-labelledby="myModalLabel" aria-hidden="true" style="display:  
none;">  
  <div class="modal-dialog">  
    <div class="loginmodal-container">
```

```
<h1>Login to Your Account</h1><br>
<form>
    <input type="text" name="user" placeholder="Username">
    <input type="password" name="pass" placeholder="Password">
    <input type="submit" name="login" class="login
        loginmodal-submit" value="Login">

</form>

<div class="login-help">
    <a href="#">Register</a> - <a href="#">Forgot Password</a>
</div>
</div>
</div>
```

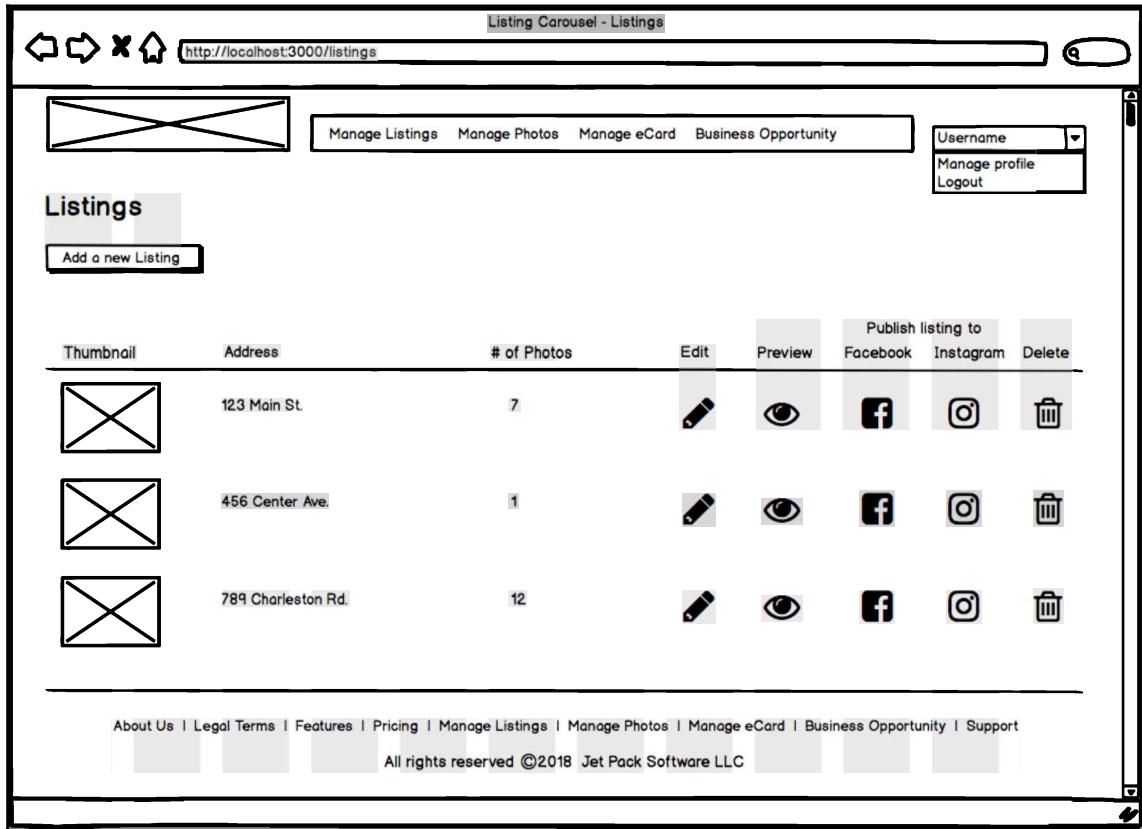
In the preceding code, we have implemented a modal window with yet another form, this time for a login feature with a number of form elements—a username and password with a submit button. Launch the page in the browser and we should see the following output:



Our application is almost taking shape now. I am sure you are as excited as I am. Let's go ahead and implement the listings page.

Listings

In previous sections, we have created our home page, signup, and login pages using the Bootstrap components. In this section, we will create our listings page. Take a look at the following wireframe. We have to loop through our listings and display a grid section, where we will display all the listings we have so far. Simple enough? You bet:



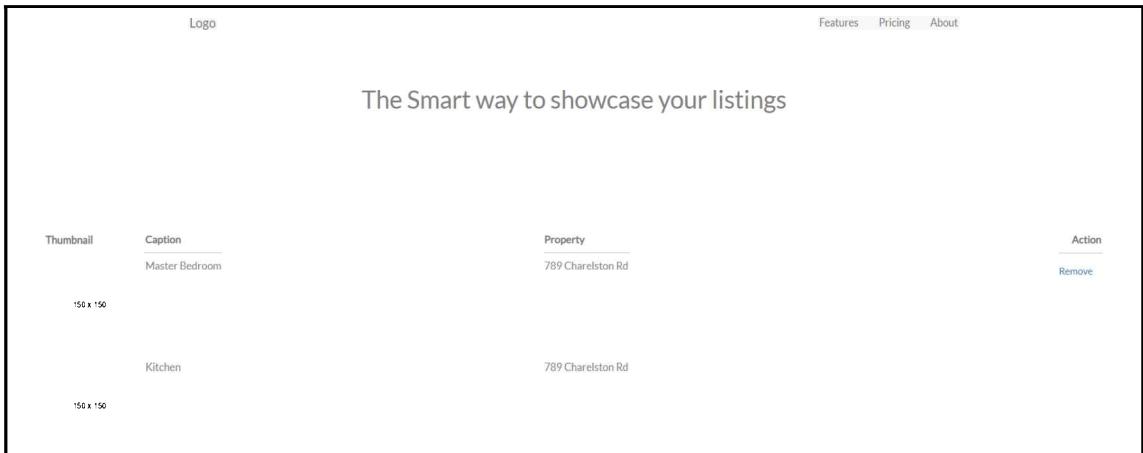
We will need to use Bootstrap's advanced layout and grid components to create the preceding layout. Take a look at the following sample code. We can achieve the preceding layout in multiple ways. We can either use the Bootstrap grid columns to design the layout or, alternatively, we can make use of table elements to design the structure. In this example, I will show you how to do it using the table elements and I will leave the grid structure to you as homework:

```
<div class="container-fluid">
  <table class="table table-hover shopping-cart-wrap">
    <thead class="text-muted">
```

```
<tr>
  <th scope="col" width="150">Thumbnail</th>
  <th scope="col">Caption</th>
  <th scope="col">Property</th>
  <th scope="col" width="200" class="text-right">Action</th>
</tr>
</thead>
<tbody>
<tr>
  <td>
    <figure class="media">
      <div class="img-wrap"><img src=
        "https://via.placeholder.com/150"
        class="img-thumbnail img-sm"></div>
    </figure>
  </td>
  <td> Master Bedroom </td>
  <td> 789 Charelston Rd </td>
  <td class="text-right"> <a title="" href="" class="btn btn-outline-success" data-toggle="tooltip" data-original-title="Save to Wishlist">
    <i class="fa fa-heart"></i></a> <a href="" class="btn btn-outline-danger">Remove</a>
  </td>
</tr>
<tr>
  <td>
    <figure class="media">
      <div class="img-wrap"><img src=
        "https://via.placeholder.com/150"
        class="img-thumbnail img-sm"></div>
    </figure>
  </td>
  <td> Kitchen </td>
  <td> 789 Charelston Rd </td>
  <td class="text-right">
    <a title="" href="" class="btn btn-outline-success" data-toggle="tooltip" data-original-title="Save to Wishlist">
      <i class="fa fa-heart"></i></a>
    <a href="" class="btn btn-outline-danger btn-round">Remove</a>
  </td>
</tr>
<tr>
  <td>
    <figure class="media">
      <div class="img-wrap"><img src=
        "https://via.placeholder.com/150"
        class="img-thumbnail img-sm"></div>
```

```
</figure>
</td>
<td> Den </td>
<td> 789 Charelston Rd </td>
<td class="text-right">
    <a title="" href="#" class="btn btn-outline-success"
        data-toggle="tooltip" data-original-title="Save to Wishlist">
        <i class="fa fa-heart"></i></a>
    <a href="#" class="btn btn-outline-danger btn-round">Remove</a>
</td>
</tr>
</tbody>
</table>
</div> <!-- card.-->
```

In the preceding code, we have created a container using the `container-fluid` class and, inside the container, we have created a table and rows structure to display our listings. In a more practical scenario, the data will always come from the backend APIs or services. For our example, and for learning purposes, we have stubbed the data here. Launch the page in the browser and we should see the output as shown in the following screenshot:



If you see the output as shown in the preceding screenshot, give yourself a pat on the back. We have made great progress in our learning. So far, we have created four pages using various different Bootstrap components and grid layouts.

In the next section, we will explore some of the other wireframes for the application, which I will leave it for you to practice. Most of the wireframes will use the same components, layouts, and grid layouts.

Create listing

In this section, I am sharing with you the wireframe for the **Create Listing** page. The Create Listing page can be easily created using the Bootstrap components and layout. Instead, we will learn to implement this using Flex-layout in the next chapter. The following is the wireframe for your reference:

The wireframe for the 'Create Listing' page is displayed in a browser window titled 'Listing Carousel - Create Listing' at the URL 'http://localhost:3000/listings/create'. The page features a header with navigation icons (refresh, back, forward, search) and a top menu bar with links for 'Manage Listings', 'Manage Photos', 'Manage eCard', and 'Business Opportunity'. A user dropdown menu shows 'Username' with a dropdown arrow, 'Manage profile', and 'Logout'. The main content area is titled 'Create Listing'. On the left, there is a vertical sidebar with input fields for 'Listing Price', 'Property type', 'Street address', 'City', 'State / Province', 'Zip / Postal code', 'Square footage', '# of bedrooms', and '# of bathrooms'. To the right of this sidebar is a large text area labeled 'Description' with a placeholder 'Enter description...'. Above the description area are two buttons: 'Select photos' and 'Upload new photo'. At the bottom right of the page is a 'Save Listing' button. The footer contains links for 'About Us', 'Legal Terms', 'Features', 'Pricing', 'Manage Listings', 'Manage Photos', 'Manage eCard', 'Business Opportunity', and 'Support', along with a copyright notice: 'All rights reserved ©2018 Jet Pack Software LLC'.

In the next section, we will see the design and wireframe details of the **Edit Listing** page.

Edit listing

In this section, we will learn about the design and wireframes for the **Edit Listing** screen. If you look carefully, the **Edit Listing** page is similar to the **Create Listing** page, except, the data is populated on load.

Again, like the **Create Listing** screen, we will design the **Edit Listing** page using Flex-layout in the next chapter.

Listing Carousel - Edit Listing (<http://localhost:3000/listings/edit>)

The wireframe shows the following layout:

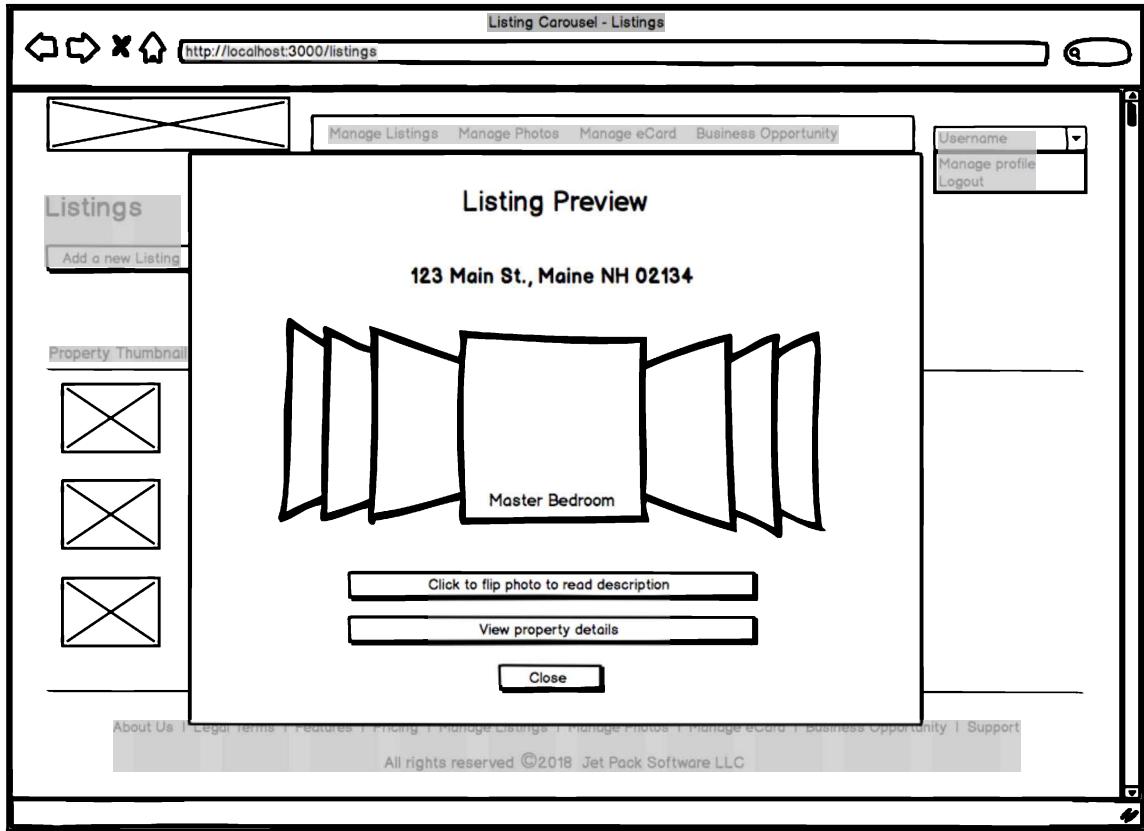
- Header:** Includes a back button, forward button, close button, and a home icon. The URL <http://localhost:3000/listings/edit> is displayed.
- Top Navigation:** Buttons for "Manage Listings", "Manage Photos", "Manage eCard", and "Business Opportunity". A dropdown menu for "Username" with options "Manage profile" and "Logout".
- Title:** "Edit Listing".
- Form Fields (Left Column):**
 - Listing Price: Input field
 - Property type: Select dropdown
 - Street address: Input field
 - City: Input field
 - State / Province: Select dropdown
 - Zip / Postal code: Input field
 - Square footage: Input field
 - # of bedrooms: Input field
 - # of bathrooms: Input field
- Form Fields (Right Column):**
 - Select additional photos: Button
 - Upload new photo: Button
 - Description: Large input area
 - Save Listing: Button
- Photos Section:** A table showing two photos:

| Thumbnail | Cover photo | Caption | Description | Edit | View | Delete |
|-----------|-------------------------------------|----------------|-------------|------|------|--------|
| | <input checked="" type="checkbox"/> | Master Bedroom | | | | |
| | | Kitchen | | | | |
- Footer:** Links to "About Us", "Legal Terms", "Features", "Pricing", "Manage Listings", "Manage Photos", "Manage eCard", "Business Opportunity", and "Support". A copyright notice: "All rights reserved ©2018 Jet Pack Software LLC".

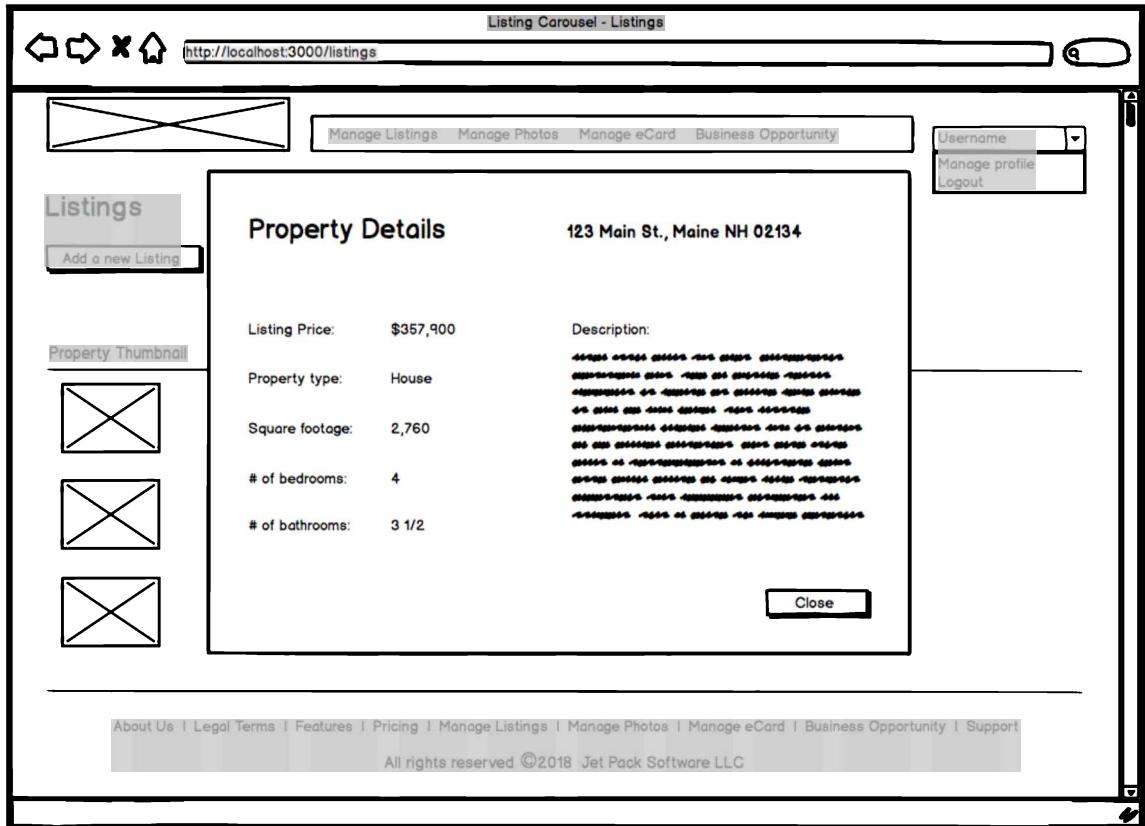
Wireframes collection

In this section, we will see the design wireframes for other pages, which we will create in chapters to come.

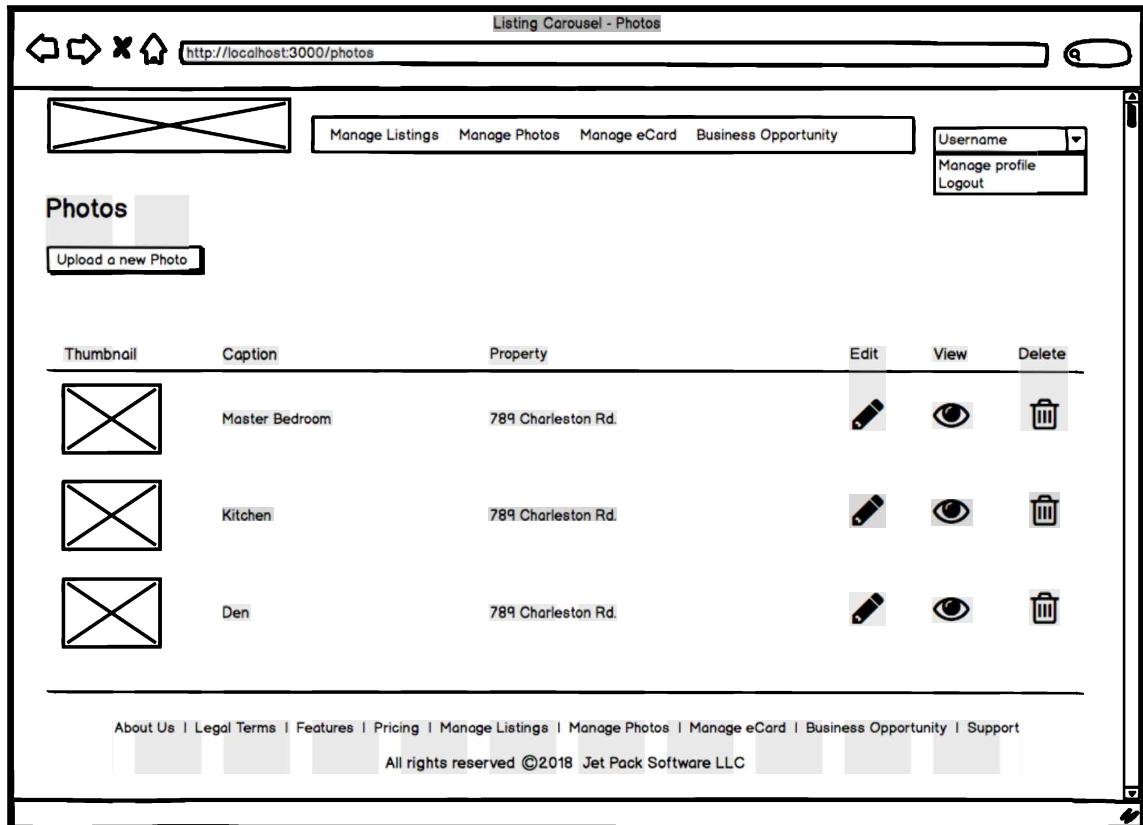
The following is the design wireframe for the **Listing Preview** page:



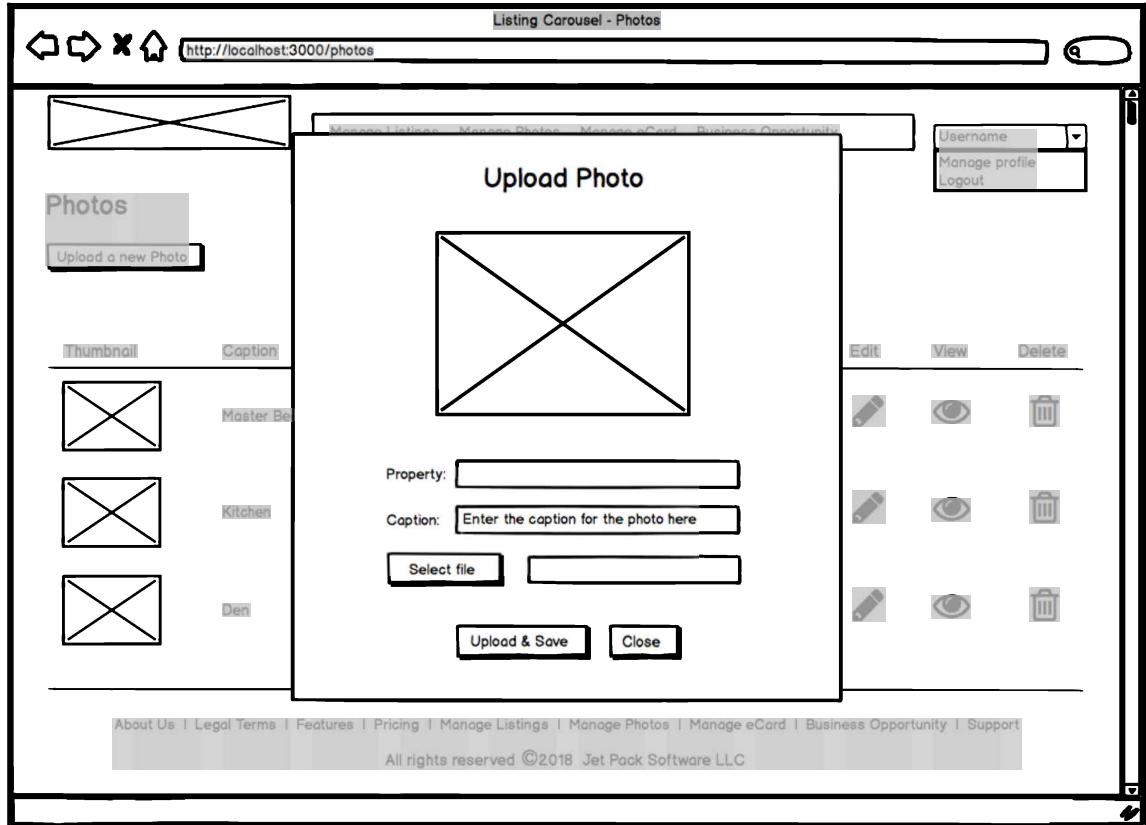
The following is the design wireframe for the property details. If you notice, we are going to use the same Bootstrap's modal window component. When we open the modal window, we should see the property details:



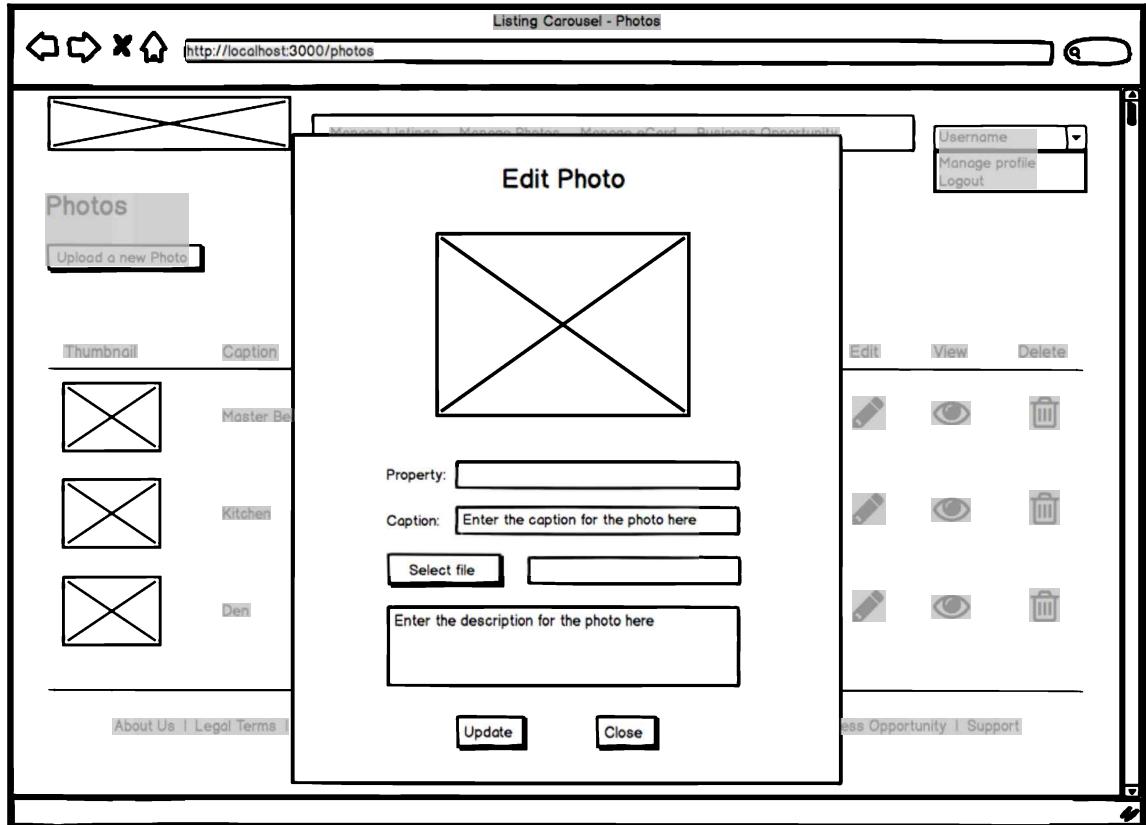
Now, we will learn how to design the wireframe for the **Photos** page. If you look carefully, the layout structure looks familiar to the **Listings** page. We will have to create a reusable design using the common libraries, which can be reused across various pages and templates:



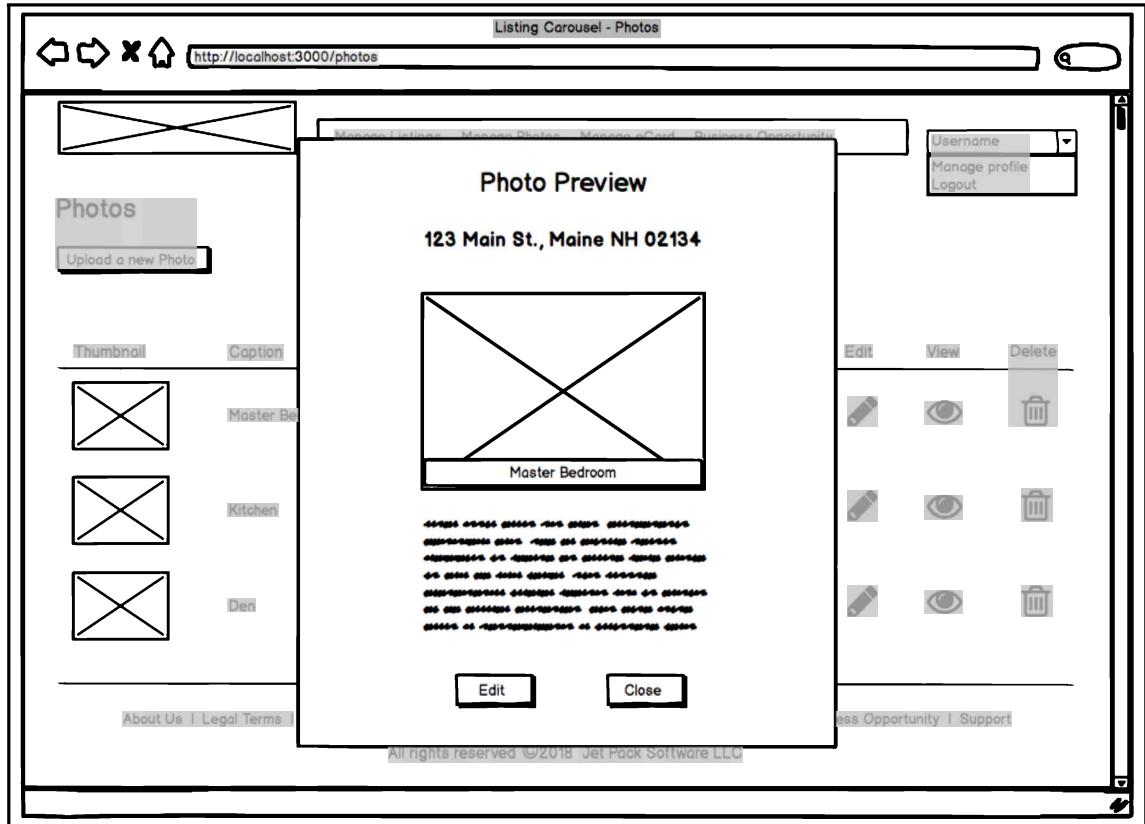
Next up is the **Upload Photo** page. We will again have to create a modal window component and provide a file upload option through which we can upload photos easily:



Now, let's move on to the **Edit Photo** wireframe. Yet again, we are making use of Bootstrap's modal window component to design our **Edit Photo** page. We will use Angular's data binding to bind the data in the modal window:



Last, but not least, we will explore the **Photo Preview** page. We can display the photos using the modal window Bootstrap component. We will close the common action buttons to close or edit the modal windows:



That was quite a bit of work we have done in this chapter, as we learned about the Bootstrap grid and layout components. We have created a few pages as part of our hands-on learning examples and designed our wireframes that we will use in our application.

Summary

This chapter was packed with all kinds of goodies. You should now understand the example application that we'll be building, the high-level game plan for the phases of our build, and the five-step process we're going to take for the first phase of our build.

We then moved on to what Sass is, and a few of its features that help us with creating our CSS for the application. We will study the tools to help you when you're writing the Sass for your applications. Next, we saw what Bootstrap is and how to integrate it into your applications. We studied what Bootstrap's grid is, and the basics for how to use it, along with some of Bootstrap's components and how to use them.

Lastly, we studied the evolution of software projects, from inception to implementation. Here, we covered the different types of analysis, the gathering of the requirements and some use cases. We also covered wireframes, went into detail on the goal of each wireframe, and the basic design principles (that were mentioned during the description of the wireframes).

So far in the book, with the exception of the quick to-do app we've built in [Chapter 1, Quick Start](#), we have not even touched Angular. This will change—starting with the next chapter, [Chapter 4, Routing](#). In this chapter, we will create the Angular shell for our application using the CLI (as we did at the beginning of the book). However, we're going to be adding routing to it. I'll explain what routing is, and how to configure routing for our application as we work through it.

So, before you turn the page, pat yourself on the back, stretch a little, and maybe pour yourself another glass of your favorite beverage. Well done, my fellow budding Angular gurus. With these first three chapters behind you, you are now ready to meet Angular!

4

Routing

The previous chapter was a monster, but it was needed to give you a couple of crash courses on two of the technologies you're likely to use, or should definitely consider using, in your web development projects (and this is true whether or not your project leverages Angular). Additionally, Chapter 3, *Bootstrap – Grid Layout and Components*, was also there to help set the stage for the rest of the book.

This chapter is much smaller by comparison, but it marks the true beginning of our journey into Angular. Each chapter from this point forward, even Chapter 12, *Integrating Backend Data Services*, where the primary focus is on building backend services in Node and MongoDB, has Angular material in it (specifically, how to use Angular's HTTP client and how to wrap your code up in an Angular service).

One other note about this chapter that I'd like to point out is that most books on Angular don't introduce routing before they introduce Angular templates and components, which is fine; but we won't be following that approach. Although routing and component templates are inextricably linked together, which is the likely reason that most books discuss routing after components, understanding components is not at all a prerequisite for understanding routing.

More generally, most programming books try and present all the material and concepts in advance, and then circle back at a later time to try and implement them in some fashion. One of the problems with this approach is that it goes against the way our brains work when assimilating and internalizing new information. It is usually better to immediately use new information, in small incremental steps.

This book focuses on being as practical as possible, as soon as possible, and in a way that maximizes the retention and understanding of new material. So, in order to accomplish this, we'll be building our example application together throughout the book, learning the topics as we need them, and not before. This means that we will often implement things that haven't yet been fully explained. They will be explained to you as we implement them, or immediately afterward—when your brain is primed, and looking for patterns to form understanding.

So, don't worry about diving in head first—it's usually the best way. I'm your guide and I am with you to all the way to the end of the book.

Here is what we'll be covering together in this chapter:

- Defining what routing is, for Angular applications
- Creating the shell of our application, as well as its first few components, using the CLI
- Configuring routing for our application
- Taking a look at route redirection, parameterized routes, and route guards
- Completing the routing configuration for our application
- Taking a look at routing strategies

There's quite a bit to cover (even for a small chapter such as this one), so let's get started!

What is routing in Angular?

Routing in Angular is simply a set of rules that map a requested URL to a component. This tends to confuse people that are coming to Angular from another technology that has routing, especially a technology that is not an SPA (that is, a single-page application) framework. Let me explain this a bit.

Angular applications only have one page (hence, the term single-page application), as we'll see in a moment when we create our Angular application. Angular components have templates, which are standard HTML elements that are used to design the layout of the structure for presentation. And as we'll see in *Chapter 6, Building Angular Components*, they also have styles.

As was mentioned in the first chapter of the book, Angular applications can be thought of as trees of components. This is to say that components can contain other components, and this nesting of components can continue as much as is required for your application.

So, although components have templates (note: some web frameworks refer to web pages as templates), Angular's routing maps URL paths to components, not to web pages or templates. When the template of the component that the requested URL is rendered (and we'll see how this happens in just a moment), not only is that component's template rendered, but all nested components' templates are also rendered. The top-level component that is mapped to by Angular's router may contain other child components, which in turn can contain other child components, and so forth. This is what is meant by a tree of components.

For the most part, data in Angular applications flow from the parent components to their immediate children. It does not flow from the parent component to its grandchild component. Moreover, data does not flow upward. It is a unidirectional flow—parent to child. I say *for the most part*, because there are techniques and libraries that change some of this behavior—for instance, components can talk to each other through an intermediary, which we'll look at later on in the book. However, by design, and without outside intervention, data flows from parent to child.

You'll become familiar with all of this as we progress through the book. All you have to understand at this point, to understand routing, is that URLs are mapped to components instead of to pages, because Angular applications only have one page. The only page in Angular apps is the `index.html` page, which is in the app directory. In Chapter 6, *Building Angular Components*, we'll see how our default component gets loaded into the `index.html` page. For now, let's get back to routing.

Creating our application's shell using the CLI

This is where it all starts. We have now reached the point where we are going to use the CLI to create our application's starting point, as well as the first bunch of components we'll need to connect them to our routing configuration. We've looked at how to install the CLI, and we've even created our first Angular application together—although our todo application was a tiny one, just to get our feet wet—back in Chapter 1, *Quick Start*.

If you haven't yet installed the CLI, you're definitely going to want to do that now. Once you've done that (hopefully, you already have), fire up your CLI, and let's begin!

The first order of business is to create a directory on your machine where you're going to place all your Angular projects. Don't create a directory for our example application, because the CLI will do that for you. Simply create a folder on your filesystem and navigate to it from your command line (if your OS is Windows), or Terminal (if your OS is a Mac or Linux). For brevity, from here on in, I'll be referring to it as your Terminal, and the folders as directories.

Next, we're going to use our CLI to create the skeleton of our application (that is, the root directory), and all the accompanying files and sub-directories that the CLI creates for us that are needed for an Angular application. Enter the following command:

```
ng new realtycarousel
```



Note: This will take about a minute to complete.

If you see **Project realtycarousel successfully created.** as the last line of output, you should now have a directory named `realtycarousel` that will contain all our application files.

The output of the preceding command is displayed in the following screenshot:

```
D:\book_2\book\chapter4_routing>ng new realtycarousel
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ http://sass-lang.com ]
CREATE  realtycarousel/angular.json (3931 bytes)
CREATE  realtycarousel/package.json (1313 bytes)
CREATE  realtycarousel/README.md (1031 bytes)
CREATE  realtycarousel/tsconfig.json (435 bytes)
CREATE  realtycarousel/tslint.json (2824 bytes)
CREATE  realtycarousel/.editorconfig (246 bytes)
CREATE  realtycarousel/.gitignore (587 bytes)
CREATE  realtycarousel/src/favicon.ico (5430 bytes)
CREATE  realtycarousel/src/index.html (301 bytes)
CREATE  realtycarousel/src/main.ts (372 bytes)
CREATE  realtycarousel/src/polyfills.ts (3571 bytes)
CREATE  realtycarousel/src/test.ts (642 bytes)
CREATE  realtycarousel/src/styles.scss (80 bytes)
CREATE  realtycarousel/src/browserlist (388 bytes)
CREATE  realtycarousel/src/karma.conf.js (980 bytes)
CREATE  realtycarousel/src/tsconfig.app.json (166 bytes)
```

Let's now test that we can run it. Navigate to your `realtycarousel` directory with the `cd` command:

```
cd realtycarousel
```

Next, start our Angular application with the CLI's server command:

```
ng serve
```

You should see a bunch of lines output to your Terminal. If one of the lines is something similar to `** NG Live Development Server is listening on localhost:4200,` open your browser on `http://localhost:4200/`, and the last line is `webpack: Compiled successfully,` then you should open a browser and point it to `http://localhost:4200`.

If you see a page with the Angular logo, this means that everything was set up correctly. You now have an empty Angular application.



You can press *Ctrl + C* to stop the CLI's development server.

Next, let's add several components, which we will reference in our routing configuration. Again, don't worry about components for now. We will look at them in depth in [Chapter 6, "Building Angular Components"](#), and [Chapter 7, "Templates, Directives, and Pipes"](#).

Run the following list of CLI commands, one at a time:

```
ng g c home
ng g c signup
ng g c login
ng g c logout
ng g c account
ng g c listings
ng g c createListing
ng g c editListing
ng g c previewListing
ng g c photos
ng g c uploadPhoto
ng g c editPhoto
ng g c previewPhoto
ng g c pageNotFound
```

The output of the first command is given in the following screenshot:

A screenshot of a terminal window showing the execution of the command 'ng g c home'. The output shows the creation of several files: 'src/app/home/home.component.html' (23 bytes), 'src/app/home/home.component.spec.ts' (614 bytes), 'src/app/home/home.component.ts' (262 bytes), and 'src/app/home/home.component.scss' (0 bytes). It also shows an update to 'src/app/app.module.ts' (467 bytes).

```
D:\book_2\book\chapter4_routing>ng new realtycarousel
D:\book_2\book\chapter4_routing>ng g c home
CREATE src/app/home/home.component.html (23 bytes)
CREATE src/app/home/home.component.spec.ts (614 bytes)
CREATE src/app/home/home.component.ts (262 bytes)
CREATE src/app/home/home.component.scss (0 bytes)
UPDATE src/app/app.module.ts (467 bytes)
```

We should see a similar output when we create all the other components.

We now have the first set of components that we need. Although their templates are empty for now, this will be good enough to enable us to configure routing for our application.

Since we'll be using Bootstrap for a few things in our application, such as its navigation bar and its responsive grid, we need to install Bootstrap along with its dependencies. In Chapter 3, *Bootstrap – Grid Layout and Components*, we simply referenced a few CDN URLs in the header of our `index.html` page in order to be able to use Bootstrap. However, we will now install Bootstrap differently—we'll be using `npm`.



You will need Node.js installed on your system in order to use the **node package manager (npm)**.

To install Bootstrap, jQuery, and Popper, run the following command in your Terminal:

```
npm install bootstrap@4 jquery popper --save
```

We have installed the libraries, and now it's time to include them in our config file so they are available throughout the application.

Open up the `angular.json` file and include the stylesheet and JavaScript files in the respective sections, as shown in the following code snippet:

```
"styles": [
  "styles.css",
  "./node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": [
  ".../node_modules/jquery/dist/jquery.min.js",
  ".../node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

The screenshot shows the edited angular.json file:

All set!

We now have the core files that we need to be able to set up routing for our application. We also made sure to install Bootstrap because we're going to create our navigation bar for our application in this chapter. Moreover, our navigation links will contain special tags that Angular uses for routing, which is another reason why we needed to install Bootstrap at this point.

Let's open our project using our IDE (again, it's easiest if you're using Visual Studio Code—but you can use whichever IDE you prefer), so we can take a look at the project structure. Additionally, in the next section, *Completing our route configuration*, we'll be making changes to a couple of files for setting things up, so you'll want to have a way to easily open and edit those files.

With the project now open in your IDE, navigate to the `app` directory, which is located within the `src` directory. As Angular developers, we'll be spending the vast majority of our time within the `app` directory. Inside the `app` directory, you will find a number of files that all start with `app`. These files make up the root component (that is, the app component) in our application, and we're going to be examining what each of these files does when we come to Chapter 6, *Building Angular Components*, where you will become very familiar with Angular components. You will see many subdirectories in the `app` directory—one for each component we created just a few moments ago, such as for `about`, `account`, `home`, and so on.

Remember, the language Angular applications are written in is TypeScript, which is what the `.ts` file extension stands for. Let's roll up our sleeves and configure routing for our application.

First things first – basic concepts

In this section, we will quickly touch base and get an overview of some of the basic concepts before we start adding routing to our Angular apps. In the basic concepts, we will learn about `BaseHref`, `RouterLink`, and `RouterLinkActive`, which we will need to implement in our templates while working with Angular routing.

Base Href

Every Angular application, in order to compose the links inside the app, should have `base href` defined at the parent level.

Open the application generated by the Angular CLI, and look inside the `index.html` file. We will see the `base href` defined to `/`, which resolves to be the root or top-level hierarchy.

The following screenshot shows the default base href configuration, as generated by the Angular CLI:

A screenshot of a code editor showing the `index.html` file. The file contains the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>RealtyCarousel</title>
  <base href="/">
```

The `<base href="/">` tag is highlighted with a yellow box. The code editor interface includes a sidebar with project files like `app-routing.module.ts`, `listing-details.component.ts`, `index.html`, etc., and a navigation bar at the top.

RouterLink and RouterLinkActive

In Chapter 7, *Templates, Directives, and Pipes*, we will learn in detail about components, directives, and templates. For now, just understand that, like the anchor element and `href` attribute in HTML5, Angular provides a way to bind the links and the URL resource:

```
<nav>
  <a routerLink="/home" routerLinkActive="active">Home</a>
  <a routerLink="/listings" routerLinkActive="active">Listings</a>
</nav>
```

In the preceding code, we are adding two links. Notice that we have added the `routerLink` attribute to the links, which will help us bind the value of `/home` and `/listings`, respectively.

Also, notice that we have added the `routerLinkActive` attribute and assigned the value as `active`. Whenever a user clicks on the link, the Angular router will know and make it active. Some call it magic!

Configuring routes for our application

It's time to add Angular routing to our application.

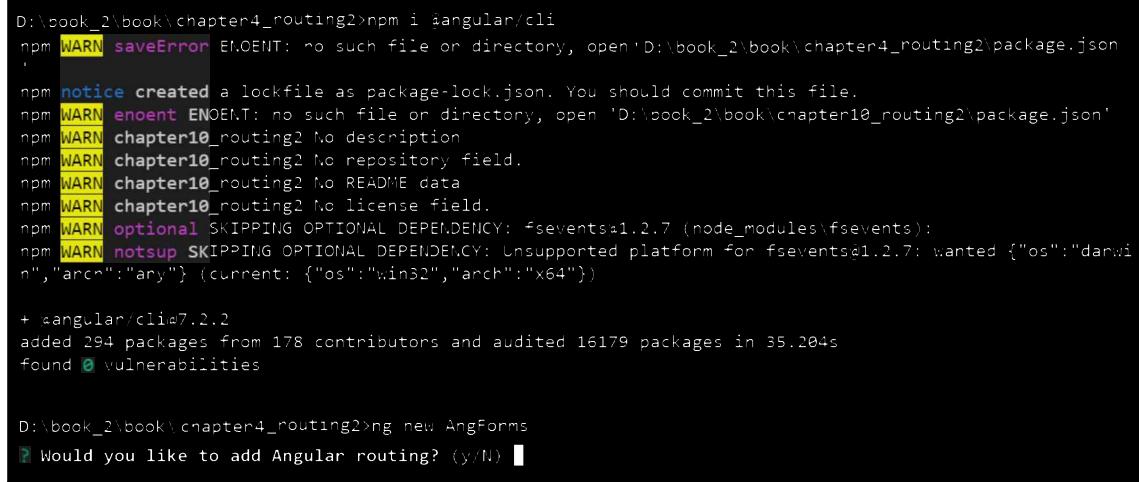
We have two options for implementing routing:

- We can use the Angular CLI to add routing during the creation of the project
- Or we can add Angular routing manually into our applications

First, let's explore the easy way, using the Angular CLI to add routing.

The Angular CLI provides us with an easy way to add routing capabilities to our Angular application. While generating a new project, the Angular CLI will prompt us to select if we want to add routing to our app.

The following screenshot shows the option displayed in the CLI for adding Angular routing:



```
D:\book_2\book\chapter4_routing2>npm i @angular/cli
npm [WARN] saveError ENOENT: no such file or directory, open 'D:\book_2\book\chapter4_routing2\package.json'
'
npm [notice] created a lockfile as package-lock.json. You should commit this file.
npm [WARN] enoent ENOENT: no such file or directory, open 'D:\book_2\book\chapter4_routing2\package.json'
npm [WARN] chapter10_routing2 No description
npm [WARN] chapter10_routing2 No repository field.
npm [WARN] chapter10_routing2 No README data
npm [WARN] chapter10_routing2 No license field.
npm [WARN] optional  SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\fsevents):
npm [WARN] notsup  SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ @angular/cli@7.2.2
added 294 packages from 178 contributors and audited 16179 packages in 35.204s
found 0 vulnerabilities

D:\book_2\book\chapter4_routing2>ng new AngForms
? Would you like to add Angular routing? (y/N) [
```

When we choose the option to add routing in our app, we are using the Angular CLI to create files, import the required modules, and create the route's rulesets.

Now, let's add routing to our project manually. Let's see how we can configure routing in our app.

In order to configure our routing, we need to follow these steps:

1. Open the `app.module.ts` file

2. Add the following `import` statement to the `import` section at the top of the file:

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

`RouterModule` contains the router service and router directives.

The `Routes` module defines the `routes` type (remember, TypeScript adds variable typing to JavaScript).

3. Write a few routes and ruleset in the `app-routing.module.ts` file:

```
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  ...
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

This code only shows three mappings:

- Mapping for `HomeComponent`
- Mapping for a redirection
- Mapping for a wildcard, or *catch-all*, the URL request

The first mapping object is the simplest case. A URL path (that is, the part of the URL that comes after the domain name) maps to one component, without any parameters (note routes can be parameterized, and we'll look at that in the *Parameterized routes* section shortly). What this route does is instruct Angular to render the `HomeComponent` template when the path in the requested URL ends in the `home`.

The second mapping object is an example of how to get one path to redirect to another URL, and, thus, route. This is typically referred to as route redirection. In our case, the path is an empty string, which means that when only the domain name is entered into the browser location bar, Angular's routing mechanism will redirect the request (that is, change the path in the URL) to `/home`. And since there is a mapping object that deals with `/home`, it will get triggered, thus rendering the `HomeComponent` template. This is a common practice for websites—entering just the domain name typically brings the user to the home or index web page. In our case, since we're building an SPA (which is what Angular web applications are), there is no home page, but rather a home component, meaning that the home component's template is rendered to simulate a home page.

The third mapping object is an example of a wildcard match and is placed as the last mapping object. When Angular's routing mechanism parses the requested URL, it compares it to the mapping objects from the top down. If the URL does not match any of the mapping rulesets, this last mapping object is triggered. What this means for our application is that, if there are no matches, the `PageNotFoundComponent` template will be rendered.

4. Now it's time to import our `appRoutes`; this is how we tell Angular about our routes. `appRoutes` is a constant that holds our route mapping, so let's create that next:

```
imports: [
  BrowserModule,
  RouterModule.forRoot(appRoutes)
]
```

5. Finally, we will need to import the `app-routing.module.ts` file into `app.module.ts`.



The complete code listing of the `app-routing.module.ts` file is in the *Completing our route configuration* section later in this chapter.

We have added the routes directly into `app.module.ts` file. It's a good practice to always separate out the route config file separately. Even better, always use the Angular CLI to add routing directly when you create the project.

That's it; we have implemented routing in our project. In the next section, we will learn in detail about adding more routes, adding parameters to our routes, and creating child routes.

Parameterized routes

Parameterized routes are routes that have a variable value as part of the URL path. For instance, a common example of this is when we reference something by its ID, as in the following:

- `/listing/23` (shows property #23 in our realty site)
- `/listing/55` (shows property #55 in our realty site)
- `/listing/721` (shows property #721 in our realty site)

Clearly, having to configure potentially hundreds of routes would not only be tedious, inefficient, and error-prone, but the maintenance of these routes (that is, removing routes and adding new ones as the inventory of property listings changed) would be troublesome.

Fortunately, Angular allows for parameterized routes, which solve issues such as these.

Take a look at the updated routes in the following code snippet:

```
const routes: Routes = [
  { path: 'home' },
  { path: 'listings/:id', component: ListingDetailsComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

If you look carefully, in the preceding routes we have added one more route, which captures the `id` of the listing, and we are also mapping it to the `ListingDetailsComponent` component.

In other words, we can also say that we have created a generic template for listings and, based on the dynamic value passed during runtime, the respective data will be displayed by the component.

That was easy. What if we have a more complex scenario that involves creating child routes? Read on.

Child routes

The routes we have created so far are very easy and straightforward use cases. In complex applications, we will need to use deep linking, which refers to hunting down a link into many levels under.

Let's take a look at some examples:

- `/home/listings` (shows listings inside home)
- `/listing/55/details` (shows the details of listing #55)
- `/listing/721/facilities` (shows the facilities of listing #721)

That's where child routes can be very handy for us to use.

In the following example, we are creating a child route inside the home route path:

```
const routes: Routes = [
  { path: 'home',
    component: HomeComponent,
    children: [
      { path: 'listings',
        component: ListingsComponent}
    ]
  },
  {path: 'listings/:id', component: ListingDetailsComponent },
  {path: '', redirectTo: '/home', pathMatch: 'full'}
];
```

In the preceding code, we are defining `children` for the `home` path and, again, we are specifying the path and component, which will correspond to the child route path.

OK, fine. This is good stuff.

What if we want to add some validation before a user can access a particular route? like a bouncer outside a club? That bouncer is called a route guard.

Route guards

As in most web applications, there are resources (that is, pages/component templates) that are accessible to everyone (such as the **Welcome page**, **Pricing page**, **About Us** page, and other informational pages), and there are other resources that are only meant to be accessed by authorized users (such as a dashboard page and an account page). That's where route guards come in, which are Angular's way to prevent unauthorized users from accessing protected parts of our application. When someone tries to access a URL that is reserved for authorized users, he will typically be redirected to the public home page of the application.

In traditional web applications, the checks and validations are implemented in the server-side code and there is practically no option to validate whether the user can access the page at the client side. But using the Angular route guard, we can implement the checks at the client side without even hitting the backend services.

The following are the various types of guards available that we can use in our applications to enhance security for authorization:

- `CanActivate`: Helps to check whether the route can be activated or not
- `CanActivateChild`: Helps to check whether the route can access child routes or not

- **CanDeactivate**: Helps to check whether the route can be deactivated or not
- **Resolve**: Helps to retrieve route data before activation of any route
- **CanLoad**: Verifies whether the user can activate the module that is being lazy loaded

And before we jump into our hands-on exercise, I want to give you a quick overview of Angular route guards, such as where to use them, how to use them, what's the return type, and so on. Route guards are always injected as a service (that is, we have `@injectable` and we will need to inject it). The guards always return a Boolean value, `true` or `false`. We can make our route guards return the observables or promises, which internally get resolved into a Boolean value.

We will continue to work on and expand the example we have created in the previous section. We are going to add a new component and call it **CRUD**. As a user, when you try to access the `crud` route, we will check when the route returns `true`. We will allow the user to navigate and see the template; otherwise, the application will throw an error prompt.

Let's dig right into the code to implement route guards. Just as we learned how to generate a component or a service, we can use the `ng` command and generate a route guard. In the Terminal, run the following command:

```
ng generate g activateAdmin
```

We have just generated a new route guard named `activateAdmin`. The output of the preceding command is displayed here:

```
D:\book_2\book\chapter4_routing2\AngularForms>ng generate g activateAdmin
CREATE src/app/activate-admin.guard.spec.ts (401 bytes)
CREATE src/app/activate-admin.guard.ts (423 bytes)
```

Let's take a look at the files generated by the Angular CLI. Open the `activate-admin.guard.ts` file in the editor. Take a look at the default code generated in the file:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from
  '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ActivateAdminGuard implements CanActivate {
  canActivate()
```

```
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean>
      | boolean {
      return true;
    }
}
```

The first few lines are just importing the required `CanActivate`, `ActivatedRouteSnapshot`, and `RouterStateSnapshot` modules from the Angular router. Next, we know that since route guards are injectable, by using the `@injectable` decorator, we are informing Angular to inject it inside the root. And we are creating a class, `ActivatedAdminGuard`, that has a method already created inside it named `canActivate`. Note that this method has to return a Boolean value, either `true` or `false`. We have created our route guard, so now let's create a route now in our `app-routing.module.ts` file.

Take a look at the updated code of the `app-routing.module.ts` file:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CrudComponent } from './crud/crud.component';
import { LoginComponent } from './login/login.component';
import { RegisterComponent } from './register/register.component';
import { ActivateAdminGuard } from './activate-admin.guard';

const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  { path: 'crud', component: CrudComponent,
  canActivate:[ActivateAdminGuard] }

];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Note that in the routes, we have added the `canActivate` interface and, for our `crud` path, when we try to launch the `crud` route, since the `canActivate` method is returning `true`, the user will be able to see the component template.

Now, go ahead and set the value to `false` and find out what happens.



If you see the application's routing go back to base href, don't be surprised.

Completing our route configuration

As promised in previous sections, I will share the entire source code of AppModule, including the route configurations. The following code may look lengthy or scary, but trust me, it's actually very simple and straightforward.

During the course of learning this chapter, we have generated many components and created their route paths. We are just importing the components and updating appRoutes with their paths. That's it. I promise.

Here is the complete listing of the app.module.ts file:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { SignupComponent } from './signup/signup.component';
import { LoginComponent } from './login/login.component';
import { ListingsComponent } from './listings/listings.component';
import { ListingDetailsComponent } from './listing-deatails/listing-
details.component';
import { EditListingComponent } from './edit-listing/edit-
listing.component';
import { PreviewListingComponent } from './preview-listing/preview-
listing.component';
import { PhotosComponent } from './photos/photos.component';
import { UploadPhotoComponent } from './upload-photo/upload-
photo.component';
import { EditPhotoComponent } from './edit-photo/edit-photo.component';
import { PreviewPhotoComponent } from './preview-photo/preview-
photo.component';
import { PageNotFoundComponent } from './page-not-found/page-not-
found.component';
import { FeaturesComponent } from './features/features.component';
import { PricingComponent } from './pricing/pricing.component';
import { AboutComponent } from './about/about.component';
import { SupportComponent } from './support/support.component';
import { AccountComponent } from './account/account.component';
import { LogoutComponent } from './logout/logout.component';
```

```
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'signup', component: SignupComponent },
  { path: 'login', component: LoginComponent },
  { path: 'logout', component: LogoutComponent },
  { path: 'account', component: AccountComponent },
  { path: 'features', component: FeaturesComponent },
  { path: 'pricing', component: PricingComponent },
  { path: 'about', component: AboutComponent },
  { path: 'support', component: SupportComponent },
  { path: 'listings', component: ListingsComponent },
  { path: 'listing/:id', component: ListingDetailsComponent },
  { path: 'listing/edit', component: EditListingComponent },
  { path: 'listing/preview', component: PreviewListingComponent },
  { path: 'photos', component: PhotosComponent },
  { path: 'photo/upload', component: UploadPhotoComponent },
  { path: 'photo/edit', component: EditPhotoComponent },
  { path: 'photo/preview', component: PreviewPhotoComponent },
  { path: '**', component: PageNotFoundComponent }
];
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    SignupComponent,
    LoginComponent,
    ListingsComponent,
    CreateListingComponent,
    EditListingComponent,
    PreviewListingComponent,
    PhotosComponent,
    UploadPhotoComponent,
    EditPhotoComponent,
    PreviewPhotoComponent,
    PageNotFoundComponent,
    FeaturesComponent,
    PricingComponent,
    AboutComponent,
    SupportComponent,
    AccountComponent,
    LogoutComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(appRoutes)
  ],
  providers: []
})
```

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

We have just created our routes, but we will need to update our template file by creating a few links that will have the path to the aforementioned-defined routes.

One thing that's most important in any application is a well-designed menu, which helps guide users and adds to a good user experience.

Using the Bootstrap `nav` component, we will design a menu for our application in the next section.

Bootstrap navigation bar and router links directives

Just before we cover a couple of routing strategies to wrap up this chapter, let's circle back and create our Bootstrap navigation bar for our application. If you recall from the previous chapter, *Chapter 3, Bootstrap – Grid Layout and Components*, I had mentioned that we'll be covering the Bootstrap navigation component in this chapter. The reason for this is that we're going to be tying our navigation bar to our routing by using routing directives as the menu links, and so the best place to cover that is in this chapter, since it falls into the domain of routing.

In the previous section, I gave you homework to enter the route path URL manually in the browser bar to see the routes working, in this section, we will add all the route URLs to the Bootstrap `navbar` component, so that the user can just click and navigate instead of typing manually.

At the starting of the chapter, we briefly touched upon `routerLink` and `routerLinkActive`. Now it's time to see them in action.

Let's take a look at the `app.component.html` file, which is the template of our app component. If you are familiar with the notion of master pages in ASP.NET, or a layout page in Rails, then you can consider the app component template as the equivalent for Angular applications. This is because the app component is the top-level component in the tree of components that will form our application. The reason I brought up the notion of a master layout is that whatever HTML is inserted into it is preserved by the server by it rendering the called page within the layout page. While this is not what happens in Angular, since it's not a server-side technology, it does hold true in concept.

What I mean by this is that whatever HTML we insert into the app component's template, it is generally still visible when other components are rendered within it. This makes the app component template a perfect place to hold our navigation bar, since it'll always be visible regardless of what component template is selected to be rendered by our routing rulesets for a given URL that is requested by our users.

Here is the code listing for our `app.component.html` file:

```
<div>
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="/">LISTCARO</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
      data-target="#navbarSupportedContent"
      aria-controls="navbarSupportedContent" aria-expanded="false"
      aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
      <ul class="navbar-nav mr-auto">
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="/" class="nav-link">Home</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="photos" class="nav-link">Photos</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="listings" class="nav-link">Listings</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="features" class="nav-link">Features</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="pricing" class="nav-link">Pricing</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="about" class="nav-link">About</a>
        </li>
        <li routerLinkActive="active" class="nav-item">
          <a routerLink="support" class="nav-link">Support</a>
        </li>
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
            role="button" data-toggle="dropdown" aria-haspopup="true"
            aria-expanded="false">
            User name
          </a>
          <div class="dropdown-menu" aria-labelledby="navbarDropdown">
```

```
<a routerLink="account" class="dropdown-item">Account</a>
<div class="dropdown-divider"></div>
<a routerLink="logout" class="dropdown-item">Log out</a>
</div>
</li>
</ul>
<form class="form-inline my-2 my-lg-0">
<button class="btn btn-outline-success my-2 my-sm-0" type="submit">
    Log In</button>
<button class="btn btn-outline-success my-2 my-sm-0" type="submit">
    Try Now</button>
</form>
</div>
</nav>
<br />
<router-outlet></router-outlet>
</div>
```

Take a deep breath, and let's analyze the preceding lines of code. We are using both Angular directives and attributes, along with Bootstrap built-in classes. So let's begin:

- We are creating a menu navbar element, `<nav>`, provided in Bootstrap, and assigning the built-in navbar classes, `navbar-expand-lg navbar-light bg-light`.
- We are also creating an element and placeholder for the logo of our application using the `navbar-brand` class.
- Using the `navbar-nav` class, we are defining a collection of links.
- We are adding few links using the anchor tag, `<a>`, and assigning the `nav-link` class, which will form the links in the menu section.
- We are also creating a drop-down menu using the `dropdown-menu` class and adding items to the menu using `dropdown-item`.
- For Angular directives and attributes, we are using `routerLink` and `routerLinkActive` and, as explained in the *First thing first - basic concepts*, section, the `routerLink` attribute is used to bind the URL resource of the link.
- To highlight the active link, we are using the `routerLinkActive` attribute. You will notice that for all links, we have assigned the attribute value as `active`. Angular at runtime will detect the link clicked and will highlight it.

Awesome, good job so far. We have implemented a nav menu for our application. We are just one step away from seeing our application in action.

Specifying the location for rendering the component templates

We need to tell Angular where we want the component templates, for the mapped components in our routing rulesets, to be displayed. For our application, we want to have the components that the router calls upon to be rendered under our navigation bar.

Angular has a directive for doing this, `<router-outlet>`, which is defined in `RouterModule`.

Under the HTML we added for creating our Bootstrap navigation bar, add this following line of HTML:

```
<router-outlet></router-outlet>
```

That's all that is needed to tell Angular where the components that are called upon by the routing service should be rendered.

Running our application

Now that we have completed configuring routing for our application, let's take it for a quick spin.

Do you remember how to build and start our Angular application? Right! Use the CLI and issue the `serve` command like this:

ng serve

Make sure you are in the application's root folder when you do this.

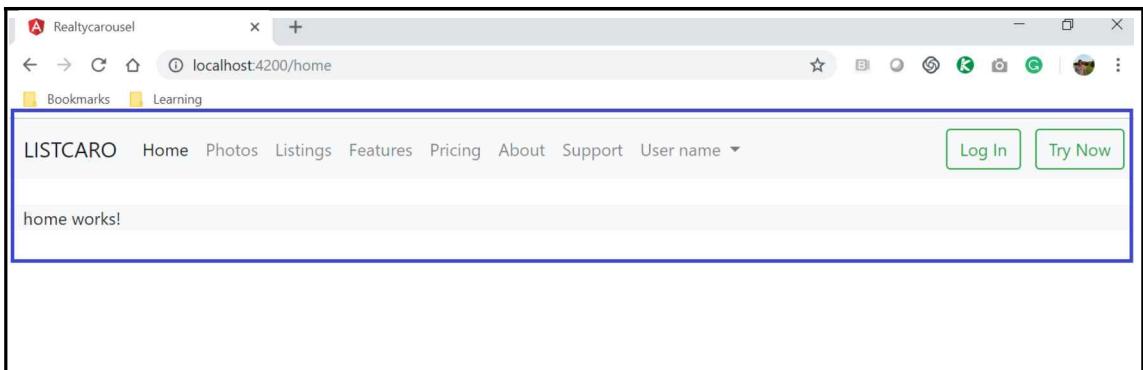


A shortcut for starting the application and opening your browser to localhost in one fell swoop, is issuing the `ng serve` command with the `--open` option, like this:

ng serve --open

What you should see is that the URL in the browser's location bar is pointing to `http://localhost:4200/home`, which is Angular routing at work. The `ng serve` command with the `open` option issued the `http://localhost:4200` URL, but this triggered the route redirection to `/home`. Pretty cool, huh?

When we run the application, we should see the output shown in the following screenshot:



In the next section, we will learn about some of the routing strategies we can implement in our apps.

Routing strategies

There are two client-side routing strategies in Angular:

- `HashLocationStrategy` (typically used for client-side purposes, such as anchor tags)
- `PathLocationStrategy` (this is the default)

To enable `HashLocationStrategy`, in the `app.module.ts` file, where we have `RouterModule.forRoot(appRoutes)`, append `{ useHash: true }` as the second parameter in the `forRoot` method. This is what it should look like:

```
RouterModule.forRoot(appRoutes, { useHash: true })
```

URLs with `HashLocationStrategy` have a hash sign (#) in their path. Here is an example:

`http://madeuplistofpeople.com/superheros#cloudman`

The preceding URL represents a get request to `http://madeuplistofpeople.com/superheros` to the server.



Everything from the hash onward is not part of the request, because the browser only sends everything in the browser's location bar, to the left of the hash sign, to the server.

The `#cloudman` portion of the URL is used exclusively by the client, and, typically, this is used by the browser to automatically scroll down to the anchor tag on the page (in this case, to the anchor tag with a `name` attribute of `cloudman`).

One use of the `HashLocationStrategy` strategy is for using the hash sign to store application state, which is convenient for implementing client-side routing for an SPA.

As an example, consider the following URLs:

- `http://madeuplistofpeople.com/#/about`
- `http://madeuplistofpeople.com/#/search`
- `http://madeuplistofpeople.com/#/contactus`

This URL pattern is great for an SPA because the only request going to the server is `http://madeuplistofpeople.com`, which is essentially one page. The client side will handle the different hash fragments (that is, from the hash sign to the end of the right-hand side of the hash sign) in whichever way it was programmed to.

To wrap up this section, an important concept of `PathLocationStrategy` is that Angular takes advantage of an HTML5 history API called `pushstate`. We can change the URL using the `pushstate` API while suppressing the traditional default action by the browser to send the new request (that is, the altered URL) to the server. This makes it possible to implement client-side routing without resorting to using a hash sign (#). This is why it is the default client-side routing strategy in Angular.

However, there is a downside. If the browser were to be refreshed, a request would be made to the server, which would reset your application with whatever the server sent back. In other words, your application would lose its state, unless you had implemented local storage strategies.

Summary

This was a fairly short chapter, but we still covered a lot of ground. In this chapter, we created the skeleton for our application, including creating the components that we had our routes mapped to. We then went through a step-by-step process of configuring routing for our application. This included importing the two required modules (that is, `RoutingModule` and `Routes`), coding up routing rulesets in the form of mapping objects, and specifying where the routed components were to be rendered.

We also installed and integrated Bootstrap into our application, and created our Bootstrap navigation bar in our root component's template. We then took a look at how to make Angular aware of installed node packages, Bootstrap and jQuery in particular, since that is how we installed Bootstrap and its dependencies (that is, jQuery and Popper).

Although we didn't make use of parameterized routes and route guards in this chapter, we mentioned them here because we'll be making use of them later in the book—in Chapter 12, *Integrating Backend Data Services*, and Chapter 14, *Advanced Angular Topics*—and in the spirit of the book, discussing things at the time we need them and not before, we are deferring their demonstration until the appropriate times.

And to wrap this chapter up, we took a look at the two client-side routing strategies that Angular lets us choose from.

We've mentioned the word *components* over and over again in this chapter, since routing maps URL paths to components. We've even created several components using the CLI, but we've spent no time understanding components. That's perfectly fine because, as was mentioned, you didn't need to understand components in order to understand routing. Now that we have routing under our belt, we'll be looking at components in the chapters ahead. But just before we do that, there is another short chapter, Chapter 5, *Flex-layout – Angular's Responsive Layout Engine*, that we're going to quickly cover. It's a bit of an odd-ball chapter because Flex-layout is an alternative to Bootstrap's responsive grid and, as such, it is not at all required to build an Angular application. However, I thought it may be of interest to you. With that said, let's turn our attention to Flex-layout.

5

Flex-Layout - Angular's Responsive Layout Engine

Flex-Layout is a TypeScript-based layout engine for Angular. It is an alternative to using Bootstrap's grid for laying out your components in your Angular projects. Flex-Layout evolved from AngularJS Material, a UI Component Framework, which was created by a team at Google led by Thomas Burleson—a well-known speaker at Angular conferences. I've not yet had the chance to attend an Angular conference, such as ng-conf or AngularMix, but I will. Maybe I'll see you there! There are many conferences on Angular spanning the globe—so you know that you're spending your time wisely studying a technology that is in high demand and is here to stay. I don't think I've said this to you yet, so I will now. Congratulations! Congratulations on selecting such a great technology to use in your projects, or maybe even as the cornerstone technology on which to build your career.

I can't help but get excited when I discover technologies that transform the way I create software for my clients and for myself, and now I get to share my excitement with you! So, pardon the slight tangent from the material at hand.

OK, let's now take a look at what we'll be covering in this chapter.

- Why this chapter was included in the book
- The four available techniques for the layout of our components
- Why FlexBox CSS is probably the best option
- What is Flex-Layout and why should you consider using it?
- Integrating Flex-Layout
- The Flex-Layout API
- Design strategies when using Flex-Layout
- Associating our wireframes and components with the chapters and topics in this book
- Implementing our selected wireframes

Why this chapter was included in the book

This is a very short chapter. In fact, it's probably the shortest in the book. However, I wanted to include it to give you options, particularly in terms of having an alternative technology to Bootstrap. Within reason, the more options you have, the better off you are. Additionally, some developers enjoy using Bootstrap, while others don't. I suspect that the reason for this is that Bootstrap's layout system is a grid. I don't know many developers who like being sandboxed into something like that. Don't get me wrong, I'm not knocking Bootstrap (Bootstrap is a great technology, and the name is even in the title of this book!), but Flex-Layout definitely feels like it is less rigid. Another reason that some developers would prefer working with something like Flex-Layout is that it feels more developer-friendly. For instance, instead of using `DIV` elements with special attributes, you use dedicated elements. This is sometimes referred to as taking a declarative approach, which sometimes simply feels more natural to developers. This may or may not make sense to you at the moment, but it will by the end of the chapter.

The four available techniques for the layout of our components

As web developers, and unless you're privileged to have a web designer on your team, we necessarily find ourselves needing to spend time with the layout of the components on our pages.

As a quick aside, let's put some terminology in place for our discussions moving forward. I've used the terms *component* and *pages* interchangeably in the first few chapters, but it is now time to get more precise. As you know, Angular applications are SPAs by default and thus only have one page. I've mentioned a few times in the book that an Angular application is like a tree of components, and it all starts with the root component.

Components are composable, which is to say that a component can be made of other components. What's the result of this? Well, we need a web page in order to have our root component rendered—and from that moment forth, our root component brings in other components, which, in turn, bring in yet other components. The end result is that our components recursively render themselves to give the illusion that we have multiple pages. Of course, we don't have multiple pages. We have a single web page, and the way in which we're architecting our application is that we have a main encompassing component for each *page* in our application. All that means is that when you see me mention *page*, instead of *component*, I'm really referring to the main component on that *page*.

Take a look back at the code we wrote for Chapter 4, *Routing*, and it should all start to be making sense to you. Specifically, a given URL maps to a component. With more traditional web applications that are not SPAs, the URLs are mapped to views, or "pages". OK, let's return our focus to considerations and available options as to layout strategies.

Laying out the components in our application includes the following four necessities:

- Laying out our components within their containers (that is, parent component and child component(s))
- Sizing our components
- Positioning our components relative to each other
- Styling the components

I don't profess to be an expert in styling or CSS. I can barely color coordinate the clothes I wear. While we have seen some CSS in Chapter 3, *Bootstrap – Grid Layout and Components*, in our SASS crash course (and we'll certainly see more CSS in the chapters ahead), this is not a book about design and styling. Packt Publishing offers a few excellent books on CSS. In this chapter, we'll only concern ourselves with laying out our components within their containers. To that end, we have four techniques that we can choose from: tables, float and clear, FlexBox CSS, and CSS Grid.

Yes, of course, Flex-Layout is also an option for us because we've chosen Angular (big smile). However, the four layout techniques I've listed apply to web development in general—regardless of a frontend framework, library, or just plain old HTML and CSS. As we have seen in Chapter 3, *Bootstrap – Grid Layout and Components*, Bootstrap is a CSS framework built on top of FlexBox CSS, and so also applies to web development in general.

Getting back to our discussion of layout techniques, let's contrast the four that are generally available to web developers, and see whether there is a clear winner. From there, we'll move on to the nitty-gritty of this chapter and take a look at what Flex-Layout is and why we should use it.

Table

Every web developer (born before the year 2000) has heard of and has likely used the TABLE tag. Where did this come from? Well, a long time ago, in a galaxy far, far away, a team of alien programmers invented HTML table tags. These aliens soon grew weary of using this layout technique, so they banned its use and banished all the web development books that taught table tags their planet. Meanwhile, somewhere on Earth, circa 1994, a web developer who was frustrated with layout issues was hit on the head with what seemed like a technical book. Its markings appeared to be some form of hieroglyphics—all indecipherable to the young techie, except the distinctly familiar markup language. The first chapter's heading was simply, <TABLE>.

Kidding aside, while tables were tremendously helpful in the early days of web development, they are now an ancient layout technique that is often frowned upon. Here are some reasons why tables are no longer the default method for laying out page elements:

- They tend to clutter up the markup in our web pages and components
- They are a maintenance nightmare, since moving things around is extremely tedious using tables
- They are rigid—more so than a grid, to the extent that we sometimes had to resort to having nested tables, which, of course, exacerbate the first two bullet points

However, despite these negatives, using tables is still a valid option and is why I've listed it here as one of the main four.

Positioning using float, and clear

CSS has some pretty cool features. Among my favorites were a couple of its declarations that deal with positioning. Specifically, the two CSS declarations I'm referring to are float and clear. These declarations can be applied to block elements, such as <div>, as well as inline elements, such as . Block-level elements are elements that occupy 100% of the parent element's space, whereas inline elements are happy to share the horizontal space they reside within their parent element.

The notion of *floating* an element (such as a `<div>`) is that it relinquishes its demand to take up the entire horizontal line. In short, it collapses its space to only consume what it needs—instead of being greedy with the horizontal real estate available—and other elements can now reside beside it, instead of being pushed below it. This is true for when the element being floated does not take up the entire space. Elements that are floated beside it, when there is not enough room horizontally, will wrap down to the next line. With that being said, you can start to see how you can achieve some degree of responsive design by using the CSS float declaration to float elements.

The purpose of *clearing* is to control how the float takes effect. When you use the CSS declaration of clear on an element, you are basically instructing that element to not float up to the higher horizontal space—even if there is space to float up in. Remember, floating elements means that the element will take the highest vertical space it can take, provided that there is room, and that its adjoining elements have also been floated (especially for block-level elements that want to consume the entire horizontal space on their own). When there isn't sufficient space, it'll wrap down to the next highest available spot, and if there is enough space, it'll float up to be beside the other element. The exception to this is if you apply the clear declaration to it in its style or class, in which it will always behave as it wraps down—even when there's room higher up. Are we good on this? Cool.

Positioning elements via *float* and *clear* definitely works, and you can create some fairly sophisticated layouts by using them. But their effects may not always be what you want to see happen, as the viewport size gets smaller. In the world of responsive layout, having as much control as possible over your layout is paramount, and being limited to just floating and clearing can often make it a bit of a challenge to have your layout rearrange itself, given a wide array of viewport sizes—at least with as much precision as the next two options give you. Another thing that you need to get used to with floating elements is that you need to reorder the listing of elements on your page, depending on whether you're floating the elements to the left or to the right.

The reason I took a little more time here on *float* and *clear* is that it's an area where too many developers don't take the time to let it sink in. The takeaway point here is that you can get very far with just this layout technique, and, depending on the nature of the project and its requirements, it may be what the doctor ordered. Of course, there's more to say about *float* and *clear* in terms of design strategies, but that is a whole other book. As always, I recommend playing around with this layout technique/strategy.

FlexBox CSS

FlexBox CSS is a layout technique that came into being with CSS3. It's a very powerful thing, and this is why other frameworks, such as Bootstrap and Flex-Layout, are built on top of it. But the best thing about FlexBox CSS is that it's understood by nearly all of the browsers in general use today. With FlexBox we get the best of both worlds — tremendous browser outreach and admirable layout flexibility for your application.

I won't say much more than this for FlexBox CSS because chances are that you will not use it—at least not directly. There are three reasons why I can probably safely make that assumption:

- Bootstrap is built on top of FlexBox CSS, and you're probably more likely to use the Bootstrap grid as opposed to using FlexBox CSS directly
- The same thing holds true with Flex-Layout, since it basically wraps FlexBox CSS in a nice API, making it easier to use

CSS Grid

CSS Grid FlexBox CSS is a layout technique that comes into being with CSS4. It's also a very powerful thing, and it makes some things easier to do than with FlexBox CSS, but, at the same time, some things are harder to implement than doing it with FlexBox CSS. Being a relatively new addition to the world of CSS, it is not as widely integrated into browsers that are generally in use.

Why FlexBox CSS is probably the best option

After reading the few preceding paragraphs in the previous section, who the winner is should come as no surprise to you. It's definitely FlexBox CSS. Let's summarize this with a list of factors that should be considered when selecting a layout option:

- **Browser reach:** As developers, we care deeply about the reach of our web applications.

- **Ease of use:** This one is a bit of a stretch, I know—both Bootstrap's grid and Flex-Layout are built on top of it, making it easier to use. But once you get your head around FlexBox CSS, most layout requirements can be handled fairly easily.
- **Ease of maintenance:** This one follows from the previous bullet point. But something that most developers are surprised by is the fact that, during the lifespan of a typical application, 20% of the time that developers are involved with it is in building it, whereas 80% of developer time with the application is in maintaining it—and so this last bullet point cannot be overemphasized.

Again, we're not considering Bootstrap and Flex-Layout to be layout techniques because they are tools/frameworks superimposed on top of the underlying layout techniques.

What is Flex-Layout and why should you use it?

We covered why the best option for us from the four for laying out our components is FlexBox CSS, but this is a chapter on Flex-Layout, and so I now need to introduce it to you. So let's do that now, and then I'll list a few reasons why you should consider using it instead of using FlexBox CSS directly (again since Flex-Layout is built on top of FlexBox CSS).

Flex-Layout's home can be found here: <https://www.github.com/angular/flex-layout>.

Here are a few bullet points detailing what Flex-Layout is:

- It's a standalone library.
- It's Angular-native (and is a TypeScript implementation).
- It's integrated with the CLI.
- It has static APIs, which are for the containers, and other static APIs, which are for the container children. These APIs have the following characteristics:
 - They are declarative
 - They support data binding and change detection
 - They are directives used in HTML
- There's no CSS for us to write since it's dynamically injected for us

Some advantages of using it instead of FlexBox CSS, and following from the preceding bullet points, are as follows:

- You don't have to be a CSS expert (in fact, as you'll soon see, we won't even be using a CSS style sheet)
- It fits Angular perfectly (in fact, it's Angular-native)
- There are APIs which helps developers in faster application development

Another nice thing to know is that since Flex-Layout is a standalone (that is, self-contained) library, it can be used with or without Angular Material. We'll be taking a look at Angular Material in Chapter 9, *Working with Angular Material*, where we'll be utilizing a couple of its components. Again, these components can be used as replacements for, or in conjunction with, ng-Bootstrap. We'll be taking a look at ng-Bootstrap in Chapter 8, *Working with NG Bootstrap*.

I've mentioned in the preceding bullet list that Flex-Layout has static APIs. What I have not mentioned is that it also has responsive APIs. We'll be covering Flex-Layout's static APIs in an upcoming section, *The Flex-Layout API*, but I leave the reading of its responsive API to you (I have included links to the Flex-Layout documentation at the end of that section).

However, I would like to say a quick word on responsive APIs. Responsive APIs are there so that you can create an adaptive UX (that is, an adaptive user experience to have a slightly different layout for various viewport sizes). In order to do that, you need to also leverage MediaQueries—not only FlexBox CSS. Now, yes, this is a chapter on Flex-Layout, so why am I mentioning that you need to leverage MediaQueries in conjunction with FlexBox CSS? I mention this to point out the fact that the Flex-Layout team has us covered in this space (that is, responsive UX, not just layout) as well. And they have done this by providing extensions to the static APIs to abstract the MediaQueries away from us. This means that we don't have to handcraft tedious rule sets—and since they created extensions on the static APIs, we can leverage what we learn there and apply the extensions to create the adaptive UX right in our HTML. It's really quite brilliant!

Integrating Flex-Layout

The Flex-Layout library comes as one self-contained module, and so we only need to import it in one place. It is more straightforward to integrate than routing was in the previous chapter.

Now let's add Flex-Layout to our project. The first thing we need to do is to install the library. In your Terminal, navigate to the root folder of the `realtycarousel` application that we started creating in Chapter 4, *Routing*, and type the following:

```
npm install --save @angular/flex-layout
```

This will install the library, so we can later import it into any of our Angular applications.

Note: If your CLI outputs a warning, such as something along the lines of "`@angular/flex-layout@5.0.0-beta.14` requires a peer of `@angular/cdk@^5.0.0` but none is installed. You must install peer dependencies yourself" (which is what happened to me), just install it the same way as anything else, as follows:

```
npm install --save @angular/cdk@^5.0.0
```

Next, we need to import it into our `RealtyCarousel` application. To do this, we need to add a couple of things to our application's main module. Open your `RealtyCarousel` project in your IDE, and then open the `app.module.ts` file from within the `src/app` directory. At the top of the file within our other import statements, add the following import statement:

```
import { FlexLayoutModule } from '@angular/flex-layout';
```

(Just underneath the `import` statement, that we added for the `RouterModule` will be just fine.)

We also need to include the `FlexLayoutModule` in the imports array within the `@NgModule` section, like this: (just underneath the `RouterModule.forRoot(appRoutes)`, we had added for the `RouterModule` would be just fine.)

With that, we're done. We now have the power of Flex-Layout at our disposal. Virtually anything else we'll do with Flex-Layout gets done in our HTML.

Let's take a look at the Flex-Layout API next—which is how we'll be leveraging Flex-Layout in our pages (that is, component templates).

The Flex-Layout API

What makes Flex-Layout easier to work with than FlexBox CSS is the fact that it has APIs that abstract the CSS away for us. We still need CSS (remember, browsers only understand HTML, JavaScript, and CSS), but what I mean by the fact that CSS will be abstracted away for us is that when our application is transpiled, Angular Flex-Layout will inject the CSS for us. As I've mentioned, Flex-Layout doesn't even have a CSS style sheet, and we don't have to write any CSS.

The following is a table of the Flex-Layout APIs, detailing what they are used for, together with a quick syntactical example:

| Type | API | Used for | Example |
|------------------------------|---------------|--|---|
| Static (for container) | fxLayout | Defines the direction of the flow (that is, flex-direction). | <div fxLayout="row" fxLayout.xs="column"></div> |
| Static (for container) | fxLayoutAlign | Defines the alignment of the elements. | <div fxLayoutAlign="start stretch"></div> |
| Static (for container) | fxLayoutWrap | Defines whether the elements should wrap. | <div fxLayoutWrap></div> |
| Static (for container) | fxLayoutGap | Sets the spacing between elements. | <div fxLayoutGap="15px"></div> |
| Static (for children) | fxFlex | Specifies resizing of the host element within its container flow layout. | <div fxFlex="1 2 calc(15em + 20px)"></div> |
| Static (for children) | fxFlexOrder | Defines the order of a FlexBox item. | <div fxFlexOrder="2"></div> |
| Static (for children) | fxFlexOffset | Offsets a FlexBox item within its container flow layout. | <div fxFlexOffset="20px"></div> |
| Static (for children) | fxFlexAlign | Similar to fxLayoutAlign, but for a specific FlexBox item (not all). | <div fxFlexAlign="center"></div> |
| Static (for children) | fxFlexFill | Maximizes dimensions of the element to that of its parent container. | <div fxFlexFill></div> |

These APIs have options, and defaults. For instance, the `fxLayout` API defaults to row, but also has column, as well as row-reverse, and column-reverse.

Also, the `.xs` in the example for the `fxLayout` API has a similar notion as the Bootstrap grid in that it provides a way to allow for different viewport sizes. So, in the first example in the preceding table, the layout for regular viewports will be that elements flow from left to right within the row, and, for small viewports, the elements will be stacked in a single column.

Another interesting thing to point out in the examples in the preceding table is where there is a calculation made in the `fxFlex` API. This is a little like what we looked at in the SASS crash course in Chapter 3, *Bootstrap – Grid Layout and Components*, although SASS was compiled by Ruby, whereas Flex-Layout is compiled by TypeScript.

I won't enumerate all the options here because you didn't purchase this book to read documentation, any more than I wrote this book to just copy documentation. Of course, I will point you to the place to look up the documentation for Flex-Layout. You can find it at their official website: <https://github.com/angular/flex-layout/wiki/API-Documentation>.

Fortunately, the Flex-Layout team has done a fantastic job with the documentation. Their wiki also includes several live layout demos that you can take a look at. Here is the direct link: <https://tburleson-layouts-demos.firebaseio.com/#/docs>.

Design strategies when using FlexBox

Since Flex-Layout is more of a flowing kind of thing, as opposed to being a grid, it's often easier to think of vertical sections of your application and assign them their own container. This is because the sections within your containers will automatically wrap downward as the viewport size gets smaller. Elements within the container should be thought of as belonging together. Contrasting this with a grid system, such as Bootstrap, the thinking is different; the cells in the grid mark the physical boundaries of the elements. The elements within the cells don't automatically wrap because when you are thinking of the design/layout, you insert elements in specific cells. Another way to conceptualize the differences between a grid and FlexBox is to think of a grid as being two-dimensional (that is, rows and columns—much like a spreadsheet), and FlexBox as being one-dimensional (in that it either flows horizontally, or it flows vertically).

Once you have your vertical containers in mind, you can then think about sub-containers flowing from left to right, and the sub-container then wrapping downward as the viewport size gets smaller—and when it does wrap downward, all the elements with that sub-container go along for the ride. Bear in mind, when I mention sub-containers, that I'm referring to the fact that FlexBox containers can be nested—which is what gives much of the layout control to the developer. When laying out your page, think of the flows as *outside-in*. This is to say that you should break up your page into large vertical sections—such as a header, main body, and footer—and then dive into each of the sections to add your sub-containers, which will flow from left to right.

It's difficult to describe *flow* in words, and so the best thing to do, as always, is to experiment with your containers and elements and study the behavior of their flow as you adjust your viewport size. This chapter includes code listings for three of our component templates (that is, *pages*) as well as their wireframes. You'll see exactly how I've designed the layouts for these component templates. Along the way, I'll also tell you why I've made some of the decisions I've made.

Associating our components with the chapters and topics in this book

Up until now, we haven't discussed where and when we'll be implementing our components. Part of the reason was that we didn't even start writing any Angular code until Chapter 4, *Routing*—with the sole exception being our to-do list mini-application in Chapter 1, *Quick Start*. However, now that we have started writing Angular code, it's time to do that now.

A good place to start the discussion is to select which component templates we'll be laying out with Flex-Layout. Since this book focuses more on Bootstrap than it does with Flex-Layout, we'll be using Bootstrap's grid for the rest of our component templates, which is the majority of them.

The first thing we'll do is to list our wireframes, as a reference, which represent our application's *pages* (that is, component templates), and we'll select three of them, which we will implement in the following section, *Implementing our selected wireframes*. And then, we'll look at the table that follows, which will show you which component templates we will implement and which chapters and, specifically, which topics we're pairing them up with.

The following is a list of our 13 wireframes from [Chapter 1, Quick Start](#):

- Home
- Sign Up
- Log In
- Edit Profile (excluded from coverage in the book)
- Property Listing (excluded from coverage in the book)
- Create Listing
- Edit Listing
- Preview Listing
- Property Details (excluded from coverage in the book)
- Photo Listing
- Upload Photo / Create Card
- Edit Photo (excluded from coverage in the book)
- Preview Photo

The following is a table of the wireframes that we will implement together in this book, along with a list of their associated chapters and topics. You can use this as a kind of roadmap in conceptually piecing together our application in your mind—meaning, at a high level, you'll know in which chapter we'll be implementing various parts of the component templates in our application:

| Wireframe /component template | Associated chapters | Associated topics |
|----------------------------------|---------------------|---|
| Home | 3 | Bootstrap grid |
| Sign Up | 3, 8, 10 | Modal dialog, ng-Bootstrap (input boxes), Forms |
| Log In | 14 | Authentication |
| Create Listing | 5, 14 | Flex-Layout, Custom validation |
| Edit Listing | 5, 10 | Flex-Layout, Forms |
| Preview Listing | 5, 6, 9 | Flex-Layout, Components, Angular Material (chips) |
| Photo Listing | 6, 7 | Components, Templates |
| Upload Photo / Create Photo Card | 10 | Forms |
| Preview Photo | 6, 9 | Components, Angular Material (card) |

The preceding table shows us topics from which chapters we will be implementing in our wireframes (that is, component templates). For example, by taking a look at the fourth row from the top, we can see that when we implement our Create Listing wireframe (that is, our `CreateListingComponent`), we'll be using Flex-Layout from this chapter, and Custom Validation from Chapter 14, *Advanced Angular Topics*.

Keep in mind that every wireframe will require components—despite not listing Chapter 6, *Building Angular Components*, in the associated chapters column, and components in the associated topics column. The reason I did this for some of the wireframes, such as Photo Listing and Preview Photo, is that we're going to discuss components a little more than we will, say, for the Sign-Up or Edit Listing wireframes. Also, certain wireframes will have us focusing much more on other topics. For instance, you can see for the Upload Photo wireframe that we'll be focusing on forms, from Chapter 10, *Working with Forms*.

Since we won't be jumping from chapter to chapter, this means that we'll be revisiting the majority of our pages (that is, component templates) two, three, or even four times, as we progress through the book.

Implementing our selected wireframes

The three wireframes (that is, component templates) that I've selected to implement with you in this chapter are the following:

- Create Listing (included because there are many sections and elements within the view)
- Edit Listing (included for the same reason as Create Listing)
- Preview Listing (included because there are very few elements within the view)

In the listing of the aforementioned wireframes , you may have noticed that there are three wireframes that have been marked as *excluded from coverage in the book*. Here is the wireframe exclusion list, along with the reason for its exclusion:

- **Edit Profile:** This has been excluded because it is just another edit form (much like the Edit Listing screen)
- **Property Listing:** This has been excluded because it is just another listing screen (much like the Photo Listing screen)
- **Property Details:** This has been excluded because it is a static screen that is uninteresting to us, from an Angular perspective
- **Edit Photo:** This has been excluded because it's yet again just another edit form

However, don't fret. All the code for the application that we'll be building together throughout the remaining pages, including the code for the four wireframes that we won't be implemented in the book, as well as the non-UI-based code (such as the Python-based APIs in Chapter 12, *Integrating Backend Data Services*, and more), are being made available to you as a download. I have you covered.

One last noteworthy point, and then we'll get on with some Flex-Layout coding. You can tell that our application will require some wireframes to be revisited more than once so that we can complete it—that is, we'll be building our application in pieces and in what seems like a chaotic back and forth fashion. This is not because the author is off his rocker—as some of his friends would love to tell you stories that make a strong case for just the opposite—but rather, it is by design. Remember, the philosophy of this book is to maximize your effectiveness of the absorption of the material, so you embark on the journey to becoming an Angular guru as quickly as possible. To the extent possible, we will immediately implement what material we cover so that it makes immediate sense, and sticks. That's the goal—and is also why I wanted to include the preceding table (that is, associating the wireframes with the chapters and topics).

There's typically a method to my madness (wink). Let's now turn our attention to the implementation of the three wireframes for this chapter.

The Create Listing wireframe

In this section, we will bring all our knowledge and understanding together and learn to create our application pages for the Create Listing page. Take a look at the following wireframe, which we will convert into code using Flex-Layout:

The wireframe illustrates the layout of the 'Create Listing' page. At the top, there is a header bar with icons for back, forward, refresh, and search, followed by the URL 'http://localhost:3000/listings/create'. Below the header is a navigation bar with links: 'Manage Listings', 'Manage Photos', 'Manage eCard', and 'Business Opportunity'. To the right of the navigation is a user profile section with a dropdown menu for 'Username', 'Manage profile', and 'Logout'. The main content area is titled 'Create Listing'. On the left side, there is a vertical sidebar with input fields for 'Listing Price', 'Property type', 'Street address', 'City', 'State / Province', 'Zip / Postal code', 'Square footage', '# of bedrooms', and '# of bathrooms'. To the right of the sidebar are two buttons: 'Select photos' and 'Upload new photo'. Below these buttons is a large text area labeled 'Description:' with a placeholder for the listing description. At the bottom right of the page is a 'Save Listing' button. At the very bottom, there is a footer with links to 'About Us', 'Legal Terms', 'Features', 'Pricing', 'Manage Listings', 'Manage Photos', 'Manage eCard', 'Business Opportunity', and 'Support'. Below the footer, a copyright notice reads 'All rights reserved ©2018 Jet Pack Software LLC'.

The other wireframe shows that we will need a header section and a two-column layout to hold the form and input elements.

We will first create a new component in our application and call it `Create Listing`. In the component template file, let's add the following sample code to the template:

```
<h1>Create Listing</h1>
<div fxLayout="row" fxLayoutAlign="space-between">
    Logo Here
</div>

<div class="bounds">

    <div class="content" fxLayout="row" class="menu">
        <div fxFlexOrder="1">Manage Listings</div>
        <div fxFlexOrder="2">Manage Photos</div>
        <div fxFlexOrder="3">Manage eCard</div>
        <div fxFlexOrder="4">Business Opportunity</div>
    </div>
    <div class="content">
        fxLayout="row"
        fxLayout.xs="column"
        fxFlexFill >
        <div fxFlex="60" class="sec1" fxFlex.xs="55">
            <form action="/action_page.php">

                <label for="lprice">Listing Price</label>
                <input type="text" id="lprice" name="lprice"
                    placeholder="Listing price">

                <label for="country">Property Type</label>
                <select id="country" name="country">
                    <option value="australia">USA</option>
                    <option value="canada">UK</option>
                    <option value="usa">UAE</option>
                </select>

                <label for="laddress">Street Address</label>
                <input type="text" id="laddress" name="laddress"
                    placeholder="Street Address">

                <label for="city">City</label>
                <input type="text" id="city" name="city" placeholder="City">

                <label for="state">State/Province</label>
                <select id="state" name="state">
                    <option value="New York">Australia</option>
                    <option value="New Jersey">Canada</option>
                </select>
            </form>
        </div>
    </div>
</div>
```

```
<option value="Texas">USA</option>
</select>

<label for="PCODE">Postal Code</label>
<input type="text" id="PCODE" name="PCODE"
placeholder="postal code">

<label for="SFOOT">Square Foot</label>
<input type="text" id="SFOOT" name="SFOOT"
placeholder="Square Foot">

<label for="BEDROOMS"># Bedrooms</label>
<input type="text" id="BEDROOMS" name="BEDROOMS"
placeholder="Bedrooms">
<label for="BATHROOMS"># Bathrooms</label>
<input type="text" id="BATHROOMS" name="BATHROOMS"
placeholder="bathrooms">

<input type="submit" value="Submit">
</form>
</div>
<div fxFlex="40" class="sec2" >

    <label for="LDESCRIPTION">Listing Description</label>
    <textarea id="LDESCRIPTION" name="LDESCRIPTION"
placeholder="Listing price"></textarea>
</div>
</div>
</div>
```

In the preceding code, we are creating a row using the `fxLayout` to create a placeholder for our logo. Next, we are creating the menu links and, using `fxFlexOrder`, we are sorting the menu links. Now, we will need to create a two-column layout, so we are now creating child elements inside the `fxLayout` row with two divisions, each `fxFlex` for 60 and 40, respectively. Inside the two columns, we will place our form input elements to create the form, as shown in the wireframe. Run the app and we should see the output, as shown in the following screenshot:

Create Listing

Logo Here

Manage Listings Manage Photos Manage eCard Business Opportunity

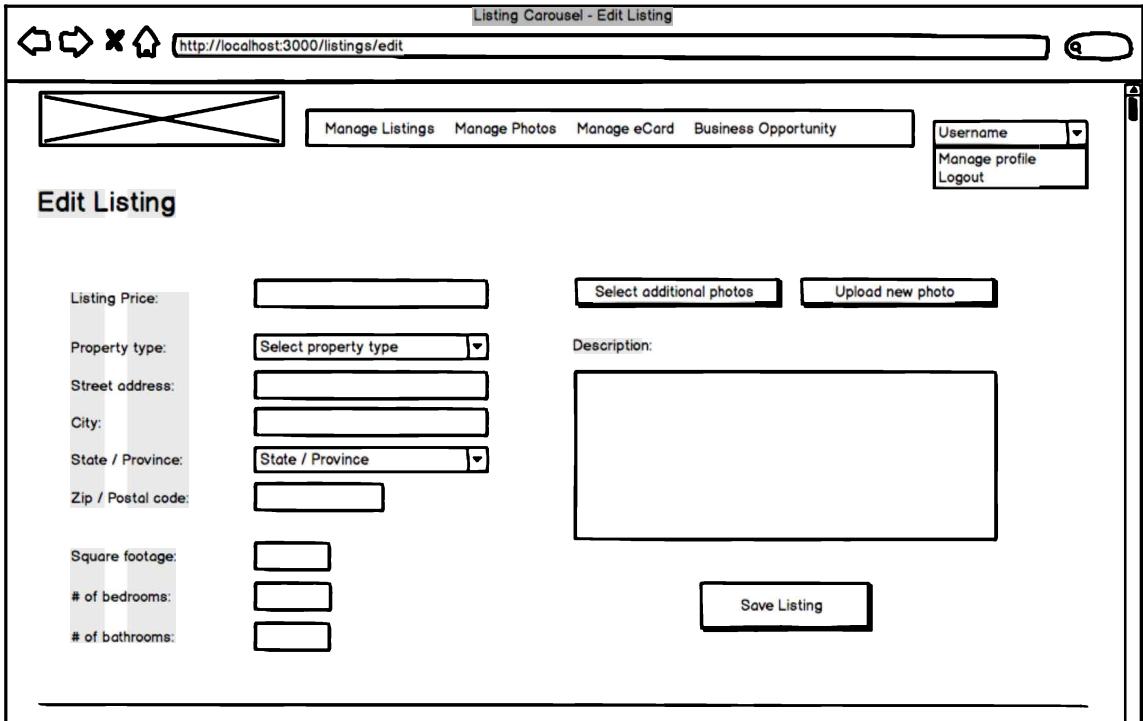
| LISTING PRICE | Listing Description |
|---|--|
| <input type="text" value="Listing price"/> | <input type="text" value="Listing price"/> |
| PROPERTY TYPE | |
| <input type="text" value="USA"/> | <input type="button" value="▼"/> |
| STREET ADDRESS | |
| <input type="text" value="Street Address"/> | |
| CITY | |
| <input type="text" value="City"/> | |
| STATE/PROVINCE | |
| <input type="text" value="Australia"/> | <input type="button" value="▼"/> |
| POSTAL CODE | |
| <input type="text" value="postal code"/> | |
| SQUARE FOOT | |
| <input type="text" value="Square Foot"/> | |
| # BEDROOMS | |
| <input type="text" value="Bedrooms"/> | |
| # BATHROOMS | |
| <input type="text" value="bathrooms"/> | |

We have got our layout ready for the **Create Listing** page. If you look closely, our labels are not exactly next to the input fields. What needs to be updated? That's right; we need to create a child column inside the main column. Try it out by way of your homework. Now, on similar lines, we can easily implement our Edit Listing page.

The Edit Listing wireframe

In the previous section, we have created our **Create Listing** page. In this section, we will learn to implement the page layout for our **Edit Listing** page. Take a look at the example we will implement. Does it not look exactly the same as the **Create Listing** page? That's correct.

The layout of the **Create** and **Edit Listing** pages will mostly be the same, except for the fact that there is data loaded when we launch the **Edit** page, whereas in the **Create** screen, there will be no data loaded initially:



Now, it's time for some code action. We will create a new component in our Angular project named `edit-listing` and, in the component template file, we will reuse the same code to create the layout from the **Create Listing** page to quickly get the **Edit Listing** page ready:

```
<h1>Edit Listing</h1>

<div fxLayout="row" fxLayoutAlign="space-between">
  Logo Here
</div>
<div class="bounds">
  <div class="content">
    <div fxLayout="row" class="menu">
      <div fxFlexOrder="1">Manage Listings</div>
      <div fxFlexOrder="2">Manage Photos</div>
      <div fxFlexOrder="3">Manage eCard</div>
      <div fxFlexOrder="4">Business Opportunity</div>
    </div>
  </div>
</div>
```

```
<div class="content"
  fxLayout="row"
  fxLayout.xs="column"
  fxFlexFill >
<div fxFlex="60" class="sec1" fxFlex.xs="55">
  <form action="/action_page.php">
    <label for="lprice">Listing Price</label>
    <input type="text" id="lprice" name="lprice"
      placeholder="Listing price">

    <label for="country">Property Type</label>
    <select id="country" name="country">
      <option value="australia">USA</option>
      <option value="canada">UK</option>
      <option value="usa">UAE</option>
    </select>

    <label for="laddress">Street Address</label>
    <input type="text" id="laddress" name="laddress"
      placeholder="Street Address">
    <label for="city">City</label>
    <input type="text" id="city" name="city"
      placeholder="City">
    <label for="state">State/Province</label>
    <select id="state" name="state">
      <option value="New York">Australia</option>
      <option value="New Jersey">Canada</option>
      <option value="Texas">USA</option>
    </select>
    <label for="PCODE">Postal Code</label>
    <input type="text" id="PCODE" name="PCODE"
      placeholder="Postal code">

    <label for="sfoot">Square Foot</label>
    <input type="text" id="sfoot" name="sfoot"
      placeholder="Square Foot">

    <label for="bedrooms"># Bedrooms</label>
    <input type="text" id="bedrooms" name="bedrooms"
      placeholder="Bedrooms">

    <label for="bathrooms"># Bathrooms</label>
    <input type="text" id="bathrooms" name="bathrooms"
      placeholder="Bathrooms">
    <input type="submit" value="Submit">
  </form>
</div>
<div fxFlex="40" class="sec2" >
```

```
<label for="ldescription">Listing Description</label>
<textarea id="ldescription" name="ldescription"
placeholder="Listing price"></textarea>

</div>
</div>
```

In the preceding code, we are creating two rows, one for the header section and the other for the content row. Inside the content row, we are creating two child columns using `fxRow`, which will be populated with the form input field elements. The output will be exactly the same as that of the Create Listing page.

Summary

This chapter provided a quick tour of exciting technology. Of course, a small book can be written that is exclusively dedicated to FlexBox CSS and Flex-Layout, so covering it in just a few pages does not do it the justice it deserves. If there is an industry that changes rapidly, it is ours, and so alternative technologies should be mentioned—and, if the technology was exciting enough, maybe even get its own chapter—regardless of which tech book and which technologies. This was precisely the case for Flex-Layout and this book. I wanted to introduce Flex-Layout to you, in some depth.

We started the chapter with a quick review of the four options for the layout techniques, explaining why FlexBox CSS is the best choice of the four. I then introduced Flex-Layout to you and presented a few compelling reasons for why you should consider using it instead of using FlexBox. Next, we saw how to integrate Flex-Layout into our Angular project, and took a look at a few of its APIs. Lastly, we circled back to our wireframes (that is, components) and associated each of them with the chapters in this book, and then implemented the components that were associated with this chapter.

I hope you enjoyed this chapter and will make it a point to experiment with Flex-Layout in at least one of your web development projects. My prediction is that many Angular developers will choose Flex-Layout as their tool of choice for laying out their components. I was already leaning toward using Flex-Layout instead of Bootstrap's grid for my next project, for all component templates.

In the next chapter, we are going to learn all about the building blocks of any Angular applications - components. We will deep-dive to learn and create some cool stuff with Angular components. Happy Reading.

6

Building Angular Components

Since the whole of Angular is composed of several interrelated parts, it's virtually impossible to select one part of Angular as being more important than another. The removal of any one of these parts renders the whole compromised—maybe even useless. Having said that, if I had to pick one part that was really important, I'd pick components. There are several really cool things about components, such as how when we build components, we are basically also extending HTML, since we're creating custom HTML tags. Components are TypeScript classes, and as we'll see a bit later on in this chapter, the way we link our code to our custom HTML tag is via the `@Component` annotation. I'll also explain what annotations are later in this chapter.

A quick word on the terminology used from this point forward: I've used the word *parts* instead of the word *components* in order to avoid confusion, since the word *component* is an overloaded word—it has different meanings in different contexts. Additionally, I use the word *page* from a classical web application perspective, as opposed to the literal sense, when talking about a view (that is, a screen).

Angular applications contain a single root component. However, when discussing an application's screens or views, it is necessary to mention other components that act as root components for that view. For example, the sign-up screen has a root component.

Here's a bullet point list of topics that we're going to cover together:

- An Angular application as a tree of components
- The `@Component` annotation
- Properties of the `@Component` annotation
- Content projection
- Life cycle hooks
- Component interface
- Components that are needed to implement the three wireframes that are associated with this chapter

Angular application architecture – a tree of components

An Angular application is basically a tree of components. As we've learned in previous chapters, Angular is an SPA framework and thus has a single page to leverage for displaying its tree of components. We've seen that Angular has a single top-level component, called the root component, and depending on what we'd like our application to do in response to our user's actions, we have that root component load up other components. These other components (let's refer to them as *secondary root components* for now) in turn recursively render additional components. The way in which we've wired up our router in Chapter 4, *Routing*, is that we've mapped URLs to our *secondary root components*—one per *page*, which springs into view when our users click on the navigational (that is, menu) links.

What makes all this possible is that components are composable. This is to say that our components are made up of other components, and are thus nested. We can nest our components in an arbitrarily deep component hierarchy, hence the statement at the very beginning of this section, *An Angular application is basically a tree of components*.

The Angular framework takes care of recursively loading and rendering our components for us.

Architecting an Angular application

Just as is the case with most engineering projects, software projects also need to have a process for designing and architecting applications. The typical way to start is to break down whatever you're building into separate chunks of work. In Angular's vernacular, this means that we need to break down our application into separate components, each of which is responsible for certain things, such as displaying the result of a calculation, or accepting user input.

Once we have a list of components that we need to use (whether they are third-party components or custom components), we need to treat them as black boxes—or mathematical functions. Let me explain what I mean by this.

When I say we need to treat our components as black boxes, I'm suggesting that we should not let our mind be consumed with their implementation at this stage (that is, when we are simply listing them). We'll concern ourselves with building our components a little later in the chapter, but for now, treating them as black boxes is all we need to do.

When I say we need to treat our components as mathematical functions, I'm merely suggesting that we think about what the output is going to be, and what inputs are needed for the function (that is, our component). The inputs and outputs of our components make up their public interfaces. We'll be taking a closer look at component interfaces a bit later in this chapter.

Breaking up your components into sub-components

The number of components in an application, or even per page, for that matter, varies greatly. It can range from just a few to several hundred—or maybe even more. However, there is a good rule of thumb for how far you should go in breaking up a component (such as a sub-component that is the top-level component for that specific page) into sub-components. If you keep component reusability in mind, all you need to ask yourself as you break down the component into sub-components is this: "Are there two or more parts to this component that I can reuse elsewhere?" If the answer is yes, you can probably benefit from breaking it down further. If the answer is no, then you're done—no more breaking down of the component is necessary.

Let's consider a quick example, just to make this a bit less abstract. Assume that you have a listing of items on a page—one row per item—and the item is a component. Let's also assume that each item has a thumbnail image for whatever that item is. If the thumbnail can be used elsewhere, maybe on the checkout page, or on the item detail page, then that thumbnail should be its own component and a sub-component of the item component.

Zooming out a bit from the item listing example, and starting from the page view, you may take this approach to help you get started when planning your components:

- Your page header is a component
- You may have a quick-links section on the right-hand side of your page, which would be another component
- You have your main content section, taking up the majority of your screen real estate, which would also be a component
- Your page footer is also a component

From the preceding components, all of them are likely to be reusable, except for the main content section. You may desire your page header, and page footer, to be on every page within your application—and you may want to re-display the quick-links section on various pages. For these reasons, those components are probably fine as they are. No further breakdown is required. The reason you will want to break down your main content component is that it's not reusable because you're not likely to have two copies of the same page!

Component responsibilities

Angular applications that are architected well will have components that are not only reusable but have well-defined boundaries. This is to say that they have a separation of concerns. Each component does one thing and does it well. The components should be abstracted away from one another, and they should not know about each other's details (that is, implementation). The one thing that they should know about each other is how to communicate with each other. This is accomplished via their public interfaces, and we'll look at this shortly.

For now, all you need to know is that when you plan your application's components, you should list their responsibilities. That is, write down what they will be doing. Astute readers will probably see the connection between the use case diagrams and the lists of component responsibilities, since components are how users will be interacting with the application.

Annotations

Annotations are a new feature of TypeScript. They are symbols, prefixed with the @ sign, that we add to our code (that is, used to decorate our classes). Annotations can appear at the top of our class declaration, or at the top of our functions, or even on top of our class properties. What annotations do, generally speaking, is inject boilerplate code where they are attached (that is, our class, function, or properties). While we don't need annotations, since we can choose to write the boilerplate code ourselves, we're better off leveraging them because the boilerplate code shouldn't have to be written over and over again.

Additionally, by using annotations instead of handwriting the boilerplate code, not only is the drudgery removed, but we don't have error-prone code to contend with. We'll see more annotations in various chapters of the book, but let's focus on the `@Component` and `@NgModule` decorators for this chapter.

`@Component`

While annotations can appear at the top of our class declaration, or at the top of our functions, or even on top of our class properties, the `@Component` annotation will always appear at the top of our component's class declaration.

In order for the `@Component` annotation to become available to us, we have to import it like this:

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

Let's look at that line of code for just a moment. It is JavaScript—specifically, ES6. If you recall from Chapter 2, *ECMAScript and TypeScript Crash Course*, the part of the curly braces of the statement is a new construct in ES6 called *destructuring*. Also, there is no explicit path to the @angular/core module. We leave it to the CLI and TypeScript compiler to figure out where the module is, and exactly how it should be loaded and made available to the code in our class.

Properties of the `@Component` decorator

The `@Component` decorator provides us with a number of properties for the purposes of configuring our components. Let's take a look at them.

selector

The `selector` is a property of the `@Component` annotation, and its value (of type `string`) is what gives the name to our custom HTML tag. I like cars, so here's an example `car` component in code, showing its annotation, selector, and class name:

```
@Component({
  selector: 'car'
})
class CarComponent {
```

When Angular sees our custom HTML tags, `<car></car>`, it creates an instance of our `CarComponent` and will replace our custom tags with some HTML that the browser actually understands. OK, but where in our component class do we add stuff to give our component something more than a ghostly aura? The next section is the answer (that is, the `template` property).

template and templateUrl

Our poor little `car` component has no visible body just yet. This is because Angular needs to know what browser-friendly HTML to add when it renders our `car` component, and we just haven't provided that for Angular yet. The way to provide that is to use the `template` property (of type `string`) to hold the HTML that Angular will render for us after it creates the instance of the `CarComponent` class (any time it sees our custom tags, `<car></car>`). Let's rectify this by beefing up our preceding `@Component` annotation:

```
@Component({
  selector: 'car',
  template: '<h3>What production car has the fastest acceleration
```

```
    time from 0 to 60?</h3><p>Tesla </p>'  
})  
class CarComponent {  
}
```

What would happen if our component required a lot of HTML? Well, this is why we have another property that we can use, `templateUrl`. The `templateUrl` property provides us with a way to externalize our component's HTML from our component class and have it in a separate file. Your `template` property would look something like this:

```
template: 'car.html'
```

styles and stylesUrls

The `styles` property is used for what you expect it to be used for—to add styling to our component template. Just like the `template` property, the value is of type `string`. Also, because it's easiest to read CSS when spaced over multiple lines, we'll be using the back-tick character (new in ES6, and thus also available in TypeScript), which enables us to create what is known as *template literals*. Let's add the `styles` parameter to our `CarComponent` class to see how this may look:

```
@Component({  
  selector: 'car',  
  template: `<h3>What production car has the fastest acceleration  
    time from 0 to 60?</h3><p>Tesla </p>',  
  styles: [  
    `.car {  
      color: #008000;  
      font-weight: bold;  
    }  
  ]  
})  
class CarComponent {  
}
```

That's all there is to the `styles` property. I bet you can guess what the `styleUrls` property does. Yup—it works just like the `templateUrl` property. It provides us with a way to externalize our component's CSS from our component class and having it in externalized style sheets. Note that I mentioned *files*, as in the plural of *file*. The `styleUrls` property takes a value of an array of strings (as opposed to being of type `String` as what the value of the `templateUrl` property is)—thus, we can pass multiple style sheets to it if we wanted to.

So, by using a combination of the template, templateUrl, styles, and styleUrls properties, we can encapsulate the HTML (that is, our component template), and the CSS we'd like to apply to our template, within our component class—thanks to the properties that the @Component annotation makes available to us. And thanks to the selector property, we can use custom HTML tags in our components' parent templates. Are you starting to get a good feel for how all these things fit together? If not, no worries—you soon will, when we start to implement our example application's views.

View encapsulation

View encapsulation is something that is extremely convenient and very cool—as most things in Angular are—and is used to configure the scope of our CSS.

Typically, when we create (or change) a CSS class, the style is applied throughout our application and is not confined to a specific page, component, and so on. Angular gives us a level of control over this by allowing us to encapsulate (that is, restrict, or contain) our styles to the components that contain the given style sheets/CSS. This is done through another property of the @Component annotation, named encapsulation.

We can set the encapsulation of our component's styles to one of the following three possible values:

- ViewEncapsulation.Emulated: This is the default value and the effect is that our styles will remain contained to just our component. They will not affect anything else on our page(s). However, our component will still inherit, or have access to, styles that are globally accessible to our application.
- ViewEncapsulation.Native: This is basically the same thing as ViewEncapsulation.Emulated, except that we are asking Angular to block, or shield, our component from any globally defined styles. The effect is that our component will be immune from any styles that are not assigned to our @Component annotation's styles or styleUrls properties.
- ViewEncapsulation.None: This is the setting that we would use if we didn't want to control the level of CSS isolation at all. In other words, if we wanted to let our component's CSS affect other page assets, and also wanted our component to inherit globally defined CSS rulesets, this is the setting that we would use.

Is this cool or what? What a feature! If you think about it, this is one of the things that make code reuse, even between applications, not just within the same application, possible. If we want to guarantee that our component will look the same across Angular applications, regardless of any given application's styles, we can set our component's encapsulation property to ViewEncapsulation.Native and we're good to go.

Module versus NgModule

Terminology is everything because it is easy to confuse things due to their semantics. This is especially true when the language/terms in the topics in question contain overloaded words—as with Angular as the topic. For instance, we've seen that we have to be pretty specific about what we mean by the words *component* and *page*. The same thing applies to the word *module*, and so I'd like to clear something up at this point before we continue on.

As we've seen in Chapter 2, *ECMAScript and TypeScript Crash Course*, the notion of modules is new in ES6. In JavaScript, when we talk about a module, we are usually referring to a code file that we can then import into the context of our executing script, making its encapsulated functions available to our script. An Angular module, or `NgModule`, is a module consisting of more than one file—hence, it's often referred to as a package. Because we treat this `NgModule`, or package, in much the same way as a JavaScript module—by importing it—we often think of them as being equivalent, but they are not.

This chapter's focus is on components, but we will also take a look at constructing our own `NgModules` in Chapter 11, *Dependency Injection and Services*, when we encapsulate our calls to our backend APIs in one cohesive package.

Before we leave our discussion on `NgModule`, deferring further discussion to a later chapter, I'd like to at least touch upon a couple of its parameters, since `@NgModule` is another annotation that I had mentioned existed.

Properties of the `@NgModule` decorator

If you take a look at the `app.module.ts` file in the example application that we started building in Chapter 4, *Routing*, you can see that there are four parameters in the `@NgModule` annotation on our `AppModule` class. Let's take a quick look at these four and what we use them for:

- **Declarations:** This is where we list the components and directives that we will need to package in this `NgModule`.
- **Imports:** These make the exported declarations of other modules available to our `NgModule`.
- **Providers:** This is where we list services and values so that they become known to **dependency injection (DI)**. They are added to the root scope and are injected to other services or directives that have them as a dependency. We're going to cover DI in chapter 12, *Integrating Backend Data Services*.

- **Bootstrap:** This is where we list the component that we want Angular to Bootstrap upon application startup.

There can only be one `NgModule` in our application where we make use of the `Bootstrap` parameter—since the bootstrapping process starts with only one module.

Content projection

The notion of content projection provides component developers with a mechanism that can increase their reusability. In particular, I'm referring to the way in which their data is displayed (that is, rendered).

What this means is that instead of trying to create a component that has properties for each possible way, its template can be altered (which is nearly impossible) so that the developers that consume the component can vary the values of these properties to customize how it's rendered. Content projection provides a way to accomplish this with much less ceremony.

The mechanism we use is a pair of `ng-content` tags, like this: `<ng-content></ng-content>`.

We'll see this in practice in the Photo Listing page, but let me show you a contrived example for now. Let's modify the template in our `CarComponent` to the following code snippet (adding the pair of `ng-content` tags):

```
template: '<h3>What production car has the fastest acceleration time from 0 to 60?</h3><ng-content></ng-content>'
```

What this does is to enable the `CarComponent`'s parent component to project content into the `CarComponent`'s template, thereby altering the template as desired. Let's assume that instead of just showing the maker of the car in regular text, within a set of `<p>` tags, we'd like to show the make of the car in a set of `` tags.

Here is what the parent component would look like:

```
<car>
  <strong>Tesla</strong>
</car>
```

It would look like the preceding instead of the following:

```
<car></car>
```

Again, this was a contrived example. Also, the whole point of Angular is to have dynamic data but we haven't done that here. For instance, we'd have the car question and answer data bound to elements within the component template, instead of having it hardcoded (to *What production car has the fastest acceleration time from 0 to 60?*, and *Tesla*, in this case.) However, our simplified hardcoded code illustrates the notion of content projection in the most straightforward way possible—by not dynamizing the data, as we'll be doing a bit later in the book.

Projecting multiple sections

It is possible to include more than a single pair of `ng-content` tags. However, since Angular will not be able to tell which projected content has replaced which set of `ng-content` tags, we need to label the `ng-content` tags in some fashion to disambiguate them from one another.

A simple way to label, or mark, the `ng-content` tags so that the intended projected content replaces the desired set of `ng-content` tags is to identify the elements by class names. We label the tags by using an attribute of `ng-content`, named `select`. Let's extend our contrived `CarComponent` example to see how this may look with two pairs of `ng-content` tags:

```
template: '<ng-content select=".question"></ng-content><ng-content
select=".answer"></ng-content>'
```

Here is what the parent component would look like:

```
<car>
  <h3 class="question">What production car has the fastest acceleration
    time from 0 to 60?</h3>
  <span select="answer"><strong>Tesla</strong></span>
</car>
```

By using `ng-content` tags, and its `select` attribute, if you have multiple content projection targets, you can create components that are customizable by developers that consume them.

Life cycle hooks

As with almost everything that is alive, from the stars in our solar system to the flowers you may have bought to decorate your dining room table, Angular components also have a life cycle—the different stages, or phases, they go through from the moment they spring into existence to the moment they cease to exist.

We can hook into these different stages to run any code that we may want to have Angular run for us as our component moves through them. This is made possible because Angular provides us with special methods, one for each phase of the component life cycle, which Angular calls for us. All we have to do is provide the code that we want Angular to run—and the way we do that is to add functions, bearing the same name as the life cycle hooks, to our component class.

There's a set of life cycle hooks for the component, and there's also a set of life cycle hooks for its children (that is, child components). The following table enumerates the most common ones:

| Life cycle hook | Type | Invoked when... |
|-----------------------|-----------|---|
| constructor | Component | Angular creates a component as a result of new being called on a class. |
| ngOnInit | Component | A component has been fully initialized. |
| ngOnChanges | Component | A change has happened to an input property (called once per chance). |
| ngOnDestroy | Component | Angular is about to destroy the component. |
| ngAfterContentInit | Child | After content projection occurs for the component. |
| ngAfterContentChecked | Child | Angular runs its change detection algorithm on the content. |
| ngAfterViewInit | Child | A component's view has been fully initialized. |
| ngAfterViewChecked | Child | Angular runs its change detection algorithm on the view. |

Most common life cycle hooks

From the preceding eight life cycle hooks, you're most likely to use only three of them (in most cases). All three fall into the component type of life cycle hook:

- `ngOnInit`: Our initialization logic for our component will go here. You may think that the constructor is the place to add the initialization logic, but `ngOnInit` is preferable because any binding of data via our interface (that is, input properties) will have been done. This isn't the case with the constructor phase.
- `ngOnChanges`: When we want to know what input properties have changed, and to what they were changed to, this is the place to look.

- `ngOnDestroy`: This is where we insert our cleanup logic for our component (if we have any things we'd like to clean up—otherwise, we don't use it).

Here is an example of how we hook into the `ngOnInit` life cycle hook (we'll just write out some output to our console):

```
class CarComponent {  
    ngOnInit() {  
        console.log('An instance of our CarComponent has  
        been fully initialized.');//  
    }  
}
```

Component interface – inputs and outputs, and the flow of data

If you were to create a diagram of your components on a particular screen (that is, view/page), drawing arrows between them to indicate the flow of data, the arrows would point from one component's outputs to another component's inputs.

In code, as we'll see later in our implementations, the ways in which we bind outputs and inputs are in our component templates (that is, in the HTML). But to have binding in HTML, we need to create our components in code and we need to give them interfaces.

Let's take a quick look at a concrete example—it'll show how a parent component could pass data to its child component. To demonstrate this, let's first create our two components.

Here is our `DadComponent`, which will be the parent component:

```
import {Component} from '@angular/core';  
@Component({  
    selector: 'dad',  
    template: `<h1>Hello. {{message}}.</h1> <br/>  
    <son *ngFor="let name of arrSonNames"  
        [Name]="name">  
        </son>  
    `,  
})  
export class DadComponent {  
    message : string = "I'm a Dad";  
    arrSonNames = ['Justin','','Brendan'];  
}
```

Here is our `SonComponent`, which will be the child component:

```
import { Component, Input, OnInit } from '@angular/core';
@Component({
  selector: 'son',
  template: `<h2>Hi. I'm a son, and my name is {{_name}}.</h2>`
})
export class SonComponent implements OnInit {
  _name: string;
  constructor() {
    console.log("The son component was just instantiated.");
  }
  ngOnInit(){
    console.log("The son component is now fully initialized.");
  }
  @Input()
  set Name(name : string ) {
    this._name = (name && name.trim()) || "I am a son.";
  }
  get Name() {
    return this._name;
  }
}
```

There's a lot going on in this little bit of code. I won't describe what's going on in the previous code blocks. Rather, I'd like for you to study it for a few minutes and see if you can figure out what's going on. You should have enough information from previous chapters, along with some basic knowledge of JavaScript/TypeScript, and an understanding of getters and setters (as many languages have). I know you can do it—give it a try. I'll give you two hints: 1) `@Input()` is a decorator, and in this case, it creates the public interface of `SonComponent`; 2) `DadComponent` will end up creating three instances of `SonComponent`. Two of the sons will know their own name, and unfortunately, one of the sons won't know his name. What does he say? What are the names of the sons that do know their name? Can you see why three sons are created? Can you guess what would be written to the console, and how many times it would be written?

We're going to see a lot of this pattern throughout our implementations, so don't worry if it looks strange, or seems a bit complicated, and you can't answer all of the questions I've asked. This stuff should become second nature to you after a while. And yes, I will be explaining our implementation code from now on—not in excruciating detail, but in enough detail for you to understand the material at hand. For now, I just wanted you to get a feel for what this passing of data via component interfaces looks like.

Our implementation of the components for our three pages

We now have enough knowledge to implement (that is, create in code) the components that we will need for the following three pages of our example application:

- Preview Listing
- Photo Listing
- Preview Photo

To generate those components, we will make use of the Angular CLI schematics. Run the following commands and we should expect the components and required files to be auto-generated:

```
ng generate component photo-listing
ng generate component preview-listing
ng generate component preview-photo
```

Once the commands run successfully, we should see the output, as shown in the following screenshot:

```
PS D:\book_2\book\chapter6_rework\LearningComponents> ng g component previewListing
CREATE src/app/preview-listing/preview-listing.component.html (34 bytes)
CREATE src/app/preview-listing/preview-listing.component.spec.ts (685 bytes)
CREATE src/app/preview-listing/preview-listing.component.ts (305 bytes)
CREATE src/app/preview-listing/preview-listing.component.scss (0 bytes)
UPDATE src/app/app.module.ts (718 bytes)
PS D:\book_2\book\chapter6_rework\LearningComponents>
PS D:\book_2\book\chapter6_rework\LearningComponents> ng g component photoListing
CREATE src/app/photo-listing/photo-listing.component.html (32 bytes)
CREATE src/app/photo-listing/photo-listing.component.spec.ts (671 bytes)
CREATE src/app/photo-listing/photo-listing.component.ts (297 bytes)
CREATE src/app/photo-listing/photo-listing.component.scss (0 bytes)
UPDATE src/app/app.module.ts (826 bytes)
PS D:\book_2\book\chapter6_rework\LearningComponents> ng g component previewPhotos
CREATE src/app/preview-photos/preview-photos.component.html (33 bytes)
CREATE src/app/preview-photos/preview-photos.component.spec.ts (678 bytes)
CREATE src/app/preview-photos/preview-photos.component.ts (301 bytes)
CREATE src/app/preview-photos/preview-photos.component.scss (0 bytes)
UPDATE src/app/app.module.ts (938 bytes)
PS D:\book_2\book\chapter6_rework\LearningComponents> █
```

In the preceding screenshot, we can notice that the corresponding files have been generated for the component and the `app.module.ts` file has been updated with the latest components generated.

The final project structure of our application with components generated so far is as follows:

The screenshot shows a code editor interface with several files open in the left sidebar. The files include `app.module.ts`, `LearningComponent... M`, `CHAPTER6_ANGLAR.COM... 1, M`, and `LearningComponents`. In the main editor area, `p.module.ts` is open, showing the following code:<app-dad></app-dad>
<router-outlet></router-outlet>Below the editor, a terminal window is open with the command `ng generate component preview-listing` entered and its output displayed:PS D:\book\book\example_code\chapter6_angular_components\LearningComponents> ng generate component preview-listing
CREATE src/app/preview-listing/preview-listing.component.html (34 bytes)
CREATE src/app/preview-listing/preview-listing.component.spec.ts (685 bytes)
CREATE src/app/preview-listing/preview-listing.component.ts (305 bytes)
CREATE src/app/preview-listing/preview-listing.component.scss (0 bytes)
UPDATE src/app/app.module.ts (719 bytes)
PS D:\book\book\example_code\chapter6_angular_components\LearningComponents>

Summary

We've covered a lot of ground in this chapter. You may have not completely understood some of the code in the last section, and this is OK because you will get good at this stuff as we implement our pages for our example application together. Since this chapter was on components, I just wanted to show you the general structure for how to set up parent and child components, and how to pass data from the parent to the child via the child's public interface. However, you should now have a pretty good understanding of how an Angular application is just a tree of components. What the rule of thumb is for breaking up your components into sub-components and what are annotations and decorators.

We also studied what the `@Component` annotation/decorator is, what its properties are, and how to configure them. We then moved on to the `@NgModule` decorator is, and what some of its properties are, and what purpose they serve. We then studied what content projection is, and how to use it to allow other developers that consume your components to customize of their rendering.

Lastly we studied what life cycle hooks are, how to use them, and reasons why you'd want to use them. We then moved on to component interfaces are and how to create them.

Finally, we studies the implementation of the components we need for our three pages (Preview Listing, Photo Listing, and Preview Photo)

In the next chapter, *Chapter 7, Templates, Directives, and Pipes*, we're going to zoom into the template portion of components, since that's where all the data binding and rendering take place—bringing our Angular application from a bunch of 0s and 1s to life on our screens.

Angular provides many tools, in the form of directives and pipes, for us to leverage, so we can tell it how to paint on our canvas. So, turn the page over and let's learn about how we can get Angular to start putting our component paint on our application canvas to bring our application to life—which is where we'll place our components onto our three pages (Preview Listing, Photo Listing, and Preview Photo).

7

Templates, Directives, and Pipes

Templates define how your components are displayed, and laid out on your web pages. Angular provides several built-in directives that provide developers control over the display of their components—from whether to display or hide the component, to rendering it multiple times on the page. Built-in directives also provide a mechanism for binding classes and styles to your components.

In Chapter 6, *Building Angular Components*, we looked at the structure of a component and how to break down our application into a tree of components.

In this chapter, you'll learn how to have command over the display of your components within their parents' templates. Specifically, at a high level, here's what we're going to be covering together:

- Templates
- Directives
- Pipes

Templates

In the previous chapter, we've seen what component templates are and how to create them. However, so far, we've only seen static HTML. In this section, I'd like to zoom in a little and take a look at some template syntax with you that allows us to create dynamic HTML, which, of course, is one of the main goals of Angular.

Templating syntax in Angular provides us with a mechanism to make our HTML dynamic—specifically, for data binding, property binding, and event binding. We'll be taking a look at these three types of binding in this chapter. The way in which Angular gives us the power to create templates that produce dynamic HTML, or to manipulate the DOM (more on this in just a little bit), is through a set of symbols.

Here are the six basic symbols that we can use:

- {{ }} for string interpolation and one-way data binding
- [()] for two-way data binding
- # for variable declaration
- () for event binding
- [] for property binding
- * prepends structural directives, such as `ngFor`, as we'll see

Directives

The three types of the directives are: components, attribute directives, and structural directives. However we're really only going to cover two of the three types of directives—**attribute directives** and **structural directives**. The reason for this is because we've already spent an entire chapter covering the first type of directive, and that was components. That's right! Components are actually directives under cover! Specifically, and stated another way (which illustrates how components are differentiated from attribute and structural directives), components are directives that have a template. Of course, this must mean that attribute and structural directives do not have templates.

OK, so what exactly are directives, then? Let's give the term *directives* a distinct definition to get rid of any possible confusion that this terminology may be causing before we move on to discuss the next two types of directives. The definition we'll be using is this: *Angular directives are constructs that provide specific DOM manipulation operations*. DOM (or HTML DOM) is an acronym for Document Object Model, and is not an Angular thing—it is a browser thing. All modern browsers create a DOM whenever a web page is loaded, which is a tree of objects that is accessible by JavaScript. Without the DOM, Angular (and any other web framework that manipulates the DOM) would not exist.

Components, as we've seen in Chapter 6, *Building Angular Components*, fit our definition of directives because they are indeed constructs that provide specific DOM operations. Not only are their templates injected into our page (replacing their custom HTML tags), but they themselves contain data, property, and event bindings, which further manipulate the DOM.

We've now fully explained components in various ways, and we'll see them in action during the implementation of our wireframes throughout the rest of the chapters that follow.

The remaining two types of directives do not inject any HTML templates in our pages or views because they do not have any templates. However, they do manipulate the DOM—as mandated by our previous definition of directives. Let's now look at what each of these types of directives do.

Attribute directives

Attribute directives manipulate the DOM by changing the appearance or the behavior of specific DOM elements. These types of directives are surrounded by brackets, and are attributes of an HTML element. The brackets are symbols (one of the five types of symbols we listed at the beginning of this chapter), and they signal to Angular that it may need to change either the appearance or the behavior of the element for which the directive is an attribute of.

That last sentence was a mouthful, so let's take a look at a code example of the attribute directive you're most likely to use. The directive I'm referring to is named `hidden`, and will cause Angular to either show or hide its element:

```
<div [hidden]="usertype != 'admin'">  
  This element, and its contents, will be hidden for all users that are not  
  Admins.  
</div>
```

In the previous code, we've hidden the `div` element and any embedded HTML from all user types that are not admins. Here, `usertype` and `admin` are, of course, application-contextual things, and are only used as an example to illustrate what Angular can do.

More generally, the `hidden` attribute directive is associated with an expression to be evaluated. The expression must evaluate to a Boolean (that is, `true` or `false`). If the expression evaluates to `true`, Angular will hide the element from the view. Conversely, if the expression evaluates to `false`, Angular will leave it alone and it will be displayed in the view.

As I did in previous chapters, I will make sure to point you to the official documentation online. As you know by now, I'm not a fan of the approach that a lot of other IT books take, which is to regurgitate documentation. While it's unavoidable to some extent, some books fill most of their pages with it. So, I will continue to stay away from that trap and will continue to add all the value I can in better ways.

That being said, the official online documentation for attribute directives can be found at <https://angular.io/guide/attribute-directives>.

Structural directives

Structural directives manipulate the DOM by adding or removing specific DOM elements. Just as we have syntax we can use to signal to Angular that we have an attribute directive to which it needs pay attention to, with the brackets symbol, we have the equivalent for structural directives.

The syntax we use to signal to Angular that we have a structural directive it needs to pay attention to is the asterisk (*). Structural directives are prefixed with an asterisk, which signals to Angular that it may need to add or remove elements from the DOM. As I enumerated at the beginning of the chapter, the asterisk is another one of the symbols we can use in our template syntax.

NgFor

Just as we look at a code example of the attribute directive you are most likely to use, let's now take a look at a code example of the structural directive you will probably use most often—NgFor:

```
<ul>
  <li *ngFor='let car of [{"make":"Porsche", "model":"Carrera"}, {"make":"Ferrari", "model":"488 Spider"}]>
    {{ car.make }}: {{ car.model }}
  </li>
</ul>
```

The previous ngFor code example prints out the following:

```
Porsche: Carrera
Ferrari: 488 Spider
```

There are a few things I'd like to point out in the previous code; firstly, the *ngFor structural directive. Let's take a look at these in bullet-point form:

- ngFor takes an iterable, and loops through it, adding elements to the DOM
- The general form of the directive's syntax is `*ngFor="let <value> of <collection>"`
- NgFor (note the capital N) refers to the class that defines the directive

- `ngFor` (note the lower case `n`) refers to both the attribute name, as well as it being an instance of the `NgFor` class
- The rest of the structural directives follow the same casing convention as `NgFor` (see the previous two bullet points)
- We can nest `ngFor` (in much the same way that we can have nested `for` each...`i` `n` loops)

Next, the collection I've provided to the `ngFor` directive is not indicative of how we would normally pass data to the directive. I've coded it this way for brevity. The way we'd normally do this is to have the data (that is, our collection) defined within our component class and assigned to a variable, and then use that variable in the statement attached to the directive.

Accessing the index value of the iteration

We'll often be interested in having access to the index value of the iteration—maybe to grab every *n*th object, or to group things in numbers of *x*, or maybe we want to implement some kind of custom pagination. Whatever the need to read the current index value of the iteration, we can use the `index` keyword to set the index to a variable within our expression.

Here's some example code demonstrating this:

```
<ul>
  <li *ngFor="let car of cars; let i = index">
    Car #{{ i + 1 }}: {{ car.model }}
  </li>
</ul>
```

In the previous code sample, let's just assume that the `cars` collection was populated elsewhere—such as in the component class.

Also, Angular takes care of updating the index value with each iteration for us—and all we have to do is to reference it.

Note that we use `{} i + 1 {}` to output the car number. This is because, as with most arrays or iterables (in most languages, but certainly in JavaScript and TypeScript), the index is zero-based. Also, note that the expression within the double curly braces, `i + 1`, is not just a variable. In Angular, whatever is inserted within the double curly braces is evaluated. We could even insert function calls there if we wanted to.

The official online documentation for structural directives is available at <https://angular.io/guide/structural-directives>.

Built-in directives

There are a few built-in directives that we have at our disposal. Let's take a peek at these in the sections that follow:

- NgFor (we've already covered this one, as the first example of a structural directive)
- NgIf
- NgSwitch, NgCase, and NgDefault
- NgStyle
- NgClass
- NgNonBindable

NgIf

When we want to either display or remove an element from the DOM, we use the NgIf directive. We pass an expression to the directive, and it must evaluate to a Boolean. If it evaluates to `true`, the element will be displayed on the view. Conversely, if the expression evaluates to `false`, the element will be removed from the DOM.

Note that we can also bind to the `hidden` property (property binding will be described as follows) to achieve the same thing, visually, but there is a difference between the property binding method, and using the NgIf directive. The difference is that using property binding on `hidden` just hides the element, whereas using the NgIf directive physically removes the element from the DOM.

Here is what NgIf looks like in code (in the context of our car example, assume we had a `horsepower` property):

```
<ul *ngFor="let car of cars">
  <li *ngIf="car.horsepower > 350">
    The {{ car.make }} {{ car.model }} is over 350 HP.
  </li>
</ul>
```

In most traditional programming languages when there are alternate things to check for in sequence, as in a series of traditional `if`, `then`, and `else` statements, it sometimes makes more sense to use a `switch` statement (if the language supports one). Java, JavaScript, and TypeScript are examples of languages (and there are, of course, many others) that support this conditional construct. Angular gives this power to us as well, so we can be more expressive and efficient with our code.

Let's take a look at how this is accomplished in Angular in the next section.

NgSwitch, NgCase, and NgDefault

In several programming languages, such as Java, JavaScript, and TypeScript, the `switch` statement does not work in isolation. It works in concert with other statements and keywords—namely, `case` and `default`. Angular's `NgSwitch` directive works exactly the same way, in that `NgSwitch` works in concert with `NgCase` and `NgDefault`.

Let's flesh out a slightly larger example here by creating a component that will contain our car data, our styling, and our template, which makes use of `NgSwitch`, with `NgCase` and `NgDefault`:

```
@Component ({
  selector: 'car-hp',
  template: `
    <h3>Cars styled by their HP range</h3>
    <ul *ngFor="let car of cars" [ngSwitch]="car.horsepower">
      <li *ngSwitchCase="car.horsepower >= 375" class="super-car">
        {{ car.make }} {{ car.model }}
      </li>
      <li *ngSwitchCase="car.horsepower >= 200 && car.horsepower
          < 375" class="sports-car">
        {{ car.make }} {{ car.model }}
      </li>
      <li *ngSwitchDefault class="grandma-car">
        {{ car.make }} {{ car.model }}
      </li>
    </ul>
  `,
  styles: [
    .super-car {
      color:#fff;
      background-color:#ff0000;
    },
    .sports-car {
      color:#000;
      background-color:#ffa500;
    },
    .grandma-car {
      color:#000;
      background-color:#ffff00;
    }
  ],
  encapsulation: ViewEncapsulation.Native
})
```

```
class CarHorsepowerComponent {
  cars: any[] = [
    {
      "make": "Ferrari",
      "model": "Testerossa",
      "horsepower": 390
    },
    {
      "make": "Buick",
      "model": "Regal",
      "horsepower": 182
    },
    {
      "make": "Porsche",
      "model": "Boxter",
      "horsepower": 320
    },
    {
      "make": "Lamborghini",
      "model": "Diablo",
      "horsepower": 485
    }
  ];
}
```

In the preceding code, we've constructed a complete component named `CarHorsepowerComponent`. Within the parent component template, Angular will replace instances of our custom HTML element, `<car-hp>`, with the template we've created in our `CarHorsepowerComponent` (this is because we assigned `car-hp` to the `selector` property of the component annotation of our `CarHorsepowerComponent` class).

We've also included the data for the collection we're passing to the `NgFor` directive within our component class, as opposed to it being inline within the expression assigned to the `NgFor` directive, as we did in a previous example.

This was a simple example whose template iterates through our `cars` collection, and applies one of three styles to the `make` and `model` of the cars based on the current car's horsepower—and this is accomplished via the `NgSwitch`, `NgCase`, and `NgDefault` directives. Specifically, here's the result:

- If the car's horsepower is equal to or greater than 375 HP, we're going to consider it to be a supercar and will have the car's `make` and `model` rendered in white font with a red background

- If the car's horsepower is equal to or greater than 200 HP, but less than 375 HP, we're going to consider it to only be a sports car and will have the car's make and model rendered in black font with an orange background
- If the car's horsepower is anything under 200 HP, which is our *default* (or *catch-all*) case, we're going to consider it to be a car that is suitably safe for a grandmother to drive, and will have the car's make and model rendered in black font with a yellow background—because most grandmothers find the color scheme of honey bees to be attractive

Of course, the grandmother comment was for entertainment value only, and I'm not trying to intentionally offend anyone who drives a car that takes a full 8 seconds, *or more*, to accelerate from 0 to 60 MPH (wink). Truth be told, one of my cars (a 2016 Honda Civic) only has 158 HP—and believe me, I've been passed on the road going uphill by a grandmother driving an Infinity Q50. That's why I bought something more powerful within the next couple of days after that horrible experience (big smile).

One last thing I wanted to point out in this previous example is the way in which the `NgSwitch` directive was used. You'll note that I wrote it in a different format, namely `[ngSwitch]="car.horsepower"`, instead of `*ngSwitch="car.horsepower"`. This is because there is a rule that Angular imposes on us when it comes to structural directives, which is that we cannot have more than one structural directive using the asterisk symbol prepending the directive's name. To work around this, we used the property binding symbol, `[]` (a pair of square brackets).

NgStyle

The `NgStyle` directive is used for setting an element's style properties. Let's rework our previous `CarHorsepowerComponent` example, which used to demonstrate `NgSwitch`, `NgCase`, and `NgDefault`, in order to show how the same desired outcome (that is, conditionally styling elements) can be better done using `NgStyle`:

```
@Component({
  selector: 'car-hp',
  template: `
    <h3>Cars styled by their HP range</h3>
    <ul *ngFor="let car of cars">
      <li [ngStyle]="{ getCarTextStyle(car.horsepower) }" >
        {{ car.make }} {{ car.model }}
      </li>
    </ul>
  `,
  encapsulation: ViewEncapsulation.Native
```

```
})
class CarHorsepowerComponent {
  getCarTextStyle(horsepower) {
    switch (horsepower) {
      case (horsepower >= 375):
        return 'color:#fff; background-color:#ff0000;';
      case (horsepower >= 200 && horsepower < 375):
        return 'color:#000; background-color:#ffa500;';
      default:
        return 'color:#000; background-color:#ffff00;';
    }
  }
  cars: any[] = [
    {
      "make": "Ferrari",
      "model": "Testerosa",
      "horsepower": 390
    },
    {
      "make": "Buick",
      "model": "Regal",
      "horsepower": 182
    },
    {
      "make": "Porsche",
      "model": "Boxter",
      "horsepower": 320
    },
    {
      "make": "Lamborghini",
      "model": "Diablo",
      "horsepower": 485
    }
  ];
}
```

In our reworking of the original `CarHorsepowerComponent` class, we've lightened up our component template by moving the logic into a function within the class. We've removed the `styles` property of the component annotation, and instead created a function (that is, `getCarTextStyle`) to return the style text to the calling function so that we can set the correct style.

Though this is a cleaner approach, we can do even better. Since we're setting a style for the car text, we can just change the style class altogether, as opposed to passing the actual style rulesets via text.

In the next section, on `NgClass`, we'll rework our code one more time to see how this is done.

NgClass

The `NgClass` directive is similar to the `NgStyle` directive, but is used to set the style class (from the CSS rulesets with the `styles` property of the component annotation), instead of setting the style via the raw CSS rulesets.

The following code example is the best choice of the last three code examples in order to achieve what we want to do:

```
@Component({
  selector: 'car-hp',
  template: `
    <h3>Cars styled by their HP range</h3>
    <ul *ngFor="let car of cars">
      <li [ngClass]=" getCarTextStyle(car.horsepower) " >
        {{ car.make }} {{ car.model }}
      </li>
    </ul>
  `,
  styles: [
    .super-car {
      color:#fff;
      background-color:#ff0000;
    },
    .sports-car {
      color:#000;
      background-color:#ffa500;
    },
    .grandmas-car {
      color:#000;
      background-color:#ffff00;
    }
  ],
  encapsulation: ViewEncapsulation.Native
})
class CarHorsepowerComponent {
  getCarTextStyle() {
    switch (horsepower) {
      case (horsepower >= 375):
        return 'super-car';
      case (horsepower >= 200 && horsepower < 375):
        return 'sports-car';
      default:
```

```
        return 'grandmas-car';
    }
}
cars: any[] = [
{
  "make": "Ferrari",
  "model": "Testerosa",
  "horsepower": 390
},
{
  "make": "Buick",
  "model": "Regal",
  "horsepower": 182
},
{
  "make": "Porsche",
  "model": "Boxter",
  "horsepower": 320
},
{
  "make": "Lamborghini",
  "model": "Diablo",
  "horsepower": 485
}
];
}
```

Here, we've kept our `styles` property for our component annotation, we've kept the template light and clean, and our function just returns the name of the CSS class to be assigned to our `NgClass` directive.

NgNonBindable

The last directive in our list to cover is the `NgNonBindable` directive. `NgNonBindable` is used when we want Angular to ignore the special symbols in our template syntax. Why would we want to do this? Well, let's say that you and I decided to create an online Angular tutorial, and the website itself was to be coded using Angular. If we wanted to render the text `{} my_value {}` to the view, Angular would try to find the `my_value` variable within its current scope to bind the value and then insert the text in its place. Since this is not what we'd want Angular to do, we need a way to instruct Angular, *Hey, by the way, do not try and evaluate and string interpolate anything right now—just render the symbols as you would with any other normal text.*

Here is what this looks like for, say, a `span` element:

```
<p>
To have Angular perform one-way binding, and render the value of my_value
onto the view, we use the    double curly braces symbol like this: <span
ngNonBindable>{{ my_value }}</span>
</p>
```

Note the placement of the `NgNonBindable` directive within the opening `` tag. When Angular sees `ngNonBindable`, it will disregard the double curly braces and will not one-way bind anything. Instead, it will let the raw text be rendered to the view.

Data binding using the `NgModel` directive

We've seen an example of one-way data binding within our example that demonstrated how to use the `NgFor` directive. Namely, one-way data binding is done using the double curly braces symbol, `{{ }}`. The variable that we enclose within the double curly braces (such as `car.make` and `car.model` from the example) is bound one-way (that is, from the component class to the template), converted to a string, and rendered to the view. It does not allow for binding any changes back to the component class.

In order for us to implement two-way data binding, thereby also allowing the binding of changes within the view back to the component class, we have to use the `NgModel` directive.

We're going to see this in action when we implement our wireframes, but let me show you what this looks like for now. In order to use `NgModel`, we have to first import an Angular module named `FormsModule` from the `forms` package, like this:

```
import { FormsModule } from '@angular/forms';
```

Then, to use this directive, we'd have something like this:

```
<div [(ngModel)]="my_content"></div>
```

Having this code in place would not only cause the view template to display the value of `my_content` in the component class, but any changes to this `div` within the view template would be then be bound back to the component class.

Event binding

We're going to see a lot of event binding during our implementation of the example application's wireframes. To bind an event that we're interested in listening for on an element, we enclose the event name within parentheses (which are one of our special symbols that we can use in our template syntax). To do so, we assign a statement to be run when the event is triggered.

Here is an example of a JavaScript alert that will be triggered when someone clicks within the `` element:

```
<span (click)="alert('This is an example of event binding in  
Angular');"></span>
```

In the preceding code, we have attached a `click` event and we invoke an alert box with the message.

Property binding

We've seen property binding in previous examples, but for completeness, I will very briefly give another example here:

```
<p class="card-text" [hidden]="true">This text will not show.</p>
```

In this previous example, we enclose the property we want to set within square brackets (which are one of the special symbols that we can use in our template syntax). Of course, this is not that useful in this example, because I've hardcoded the Boolean to `true` instead of using an expression that is to be evaluated—but the point of this example was to focus on the `[hidden]` part.

Custom directives

Angular is extensible. Not only can we easily create our own custom components (so that we're not restricted to using ready-made components from third parties), but we can also create our own attribute directives so that we're not restricted to what Angular gives us out of the box.

I'll leave some of the custom things we can do in Angular, such as custom attribute directives, custom pipes (we'll see what pipes are in the next section), and custom form validation, until Chapter 14, *Advanced Angular Topics*. We'll be taking a look at form validation in Chapter 10, *Working with Forms*. There's a good reason why I chose to lump all of the advanced stuff that is covered in this book into one chapter—to give you time to digest the basic stuff first. By the time the advanced chapter comes along, near the end of the book, you'll be ready and primed to more easily absorb that information.

Pipes

Pipes are used for formatting the data in our template views. Pipes will take data as input and transform it into our desired format for displaying it to end user. We can use the pipe property (`|`) in any Angular template or view in our project.

Let me give you a quick rundown before we jump into creating our examples. Let's say that, from the backend service, we get the price of a product as 100, and based on the user's country or preference, we may want to display the value as \$100 if the user is from the USA, or INR 100 if the user is from India. So, we are able to transform the way that we display the price without any major complexity. This is thanks to the currency pipe operator.

Angular provides a lot of built-in pipes ready to use directly in our templates. Additionally, we can also create our own custom pipes to extend our application's functionality.

Here's the list of all the built-in pipes that Angular provides:

- Lowercase pipe
- Uppercase pipe
- Date pipe
- Currency pipe
- JSON pipe
- Percent pipe
- Decimal pipe
- Slice pipe

We will learn about each of the available built-in pipes by doing some fun, practical examples. We can make use of any of the existing template files that we have created in our Angular project so far.

We will need some data that we want to process and transform using our pipes. I am going to quickly create a dataset in our `app.component.ts` file:

```
products: any[] = [
  {
    "code": "p100",
    "name": "Moto",
    "price": 390.56
  },
  {
    "code": "p200",
    "name": "Micro",
    "price": 240.89
  },
  {
    "code": "p300",
    "name": "Mini",
    "price": 300.43
  }
];
```

We have created a sample dataset for products in our app component. Great, now we are good to apply our pipes in our `app.component.html` file. We are going to keep it simple in our template. We will just create a table and bind the values in the table. If you are feeling a little adventurous today, go ahead create a layout for our application using Flex-Layout, which we learned in Chapter 5, *Flex-Layout – Angular's Responsive Layout Engine*:

```
<h4>Learning Angular Pipes</h4>


|                  |                  |                   |
|------------------|------------------|-------------------|
| Product Code     | Product Name     | Product Price     |
| {{product.code}} | {{product.name}} | {{product.price}} |


```

In the preceding sample code, we have created a table and, using the data binding, we have bound the data in our template. Now it's time to use the pipe operator in our template. To apply any pipe, we will have to add the pipe operator against the data, as shown in the following syntax:

```
 {{ data | <pipe name> }}
```

We can easily transform our product name into uppercase by applying the uppercase pipe, as follows:

```
<td>{{product.name | uppercase }}</td>
```

Similarly, we can apply the lowercase pipe as well, which will make all characters lowercase:

```
<td>{{product.name | lowercase }}</td>
```

That was very simple, you might say? So it is! Let's keep rolling. In a similar way, we will use the number pipe operator to show or hide the decimal points.

For displaying product prices, we want to add the currency; no problem, we will use the currency pipe:

```
<td>{{product.price | currency }}</td>
```

In the preceding example, we have transformed the product price by adding the currency pipe. The remaining pipe operators I am leaving to you as homework.

By default, \$ currency is added when we use the currency pipe.



We can customize it by parameterizing the currency pipe. We will learn how to pass parameters to the pipe operators. We will have to extend the syntax of the pipe operator by passing the parameters as follows:

```
 {{ data | pipe : <parameter1 : parameter2> }}
```

The preceding syntax looks similar to how we learned to define a pipe operator, except that now it has two parameters. We can define a pipe operator with any number of parameters based on our requirements. We have used the currency operator in the previous example, so let's pass parameters to extend the currency pipe operator:

```
<td>{{ product.price | currency: 'INR' }}</td>
```

We are passing the `INR` parameter to our currency pipe operator. Now, the output of the currency pipe operator will not be `$` anymore; instead, it will be as shown in the following screenshot:

| Learning Angular Pipes | | | |
|------------------------|--------------|---------------|--|
| Product Code | Product Name | Product Price | |
| p100 | MOTO | ₹390.56 | |
| p200 | MICRO | ₹240.89 | |
| p300 | MINI | ₹300.43 | |

We have learned to use built-in pipe operators in this section. Now, we'll learn about creating our own custom pipes. .

Custom pipes

Angular is extensible in this area of custom pipes as well as well as custom directives. However, I'm deferring our discussion of custom pipes until Chapter 14, *Advanced Angular Topics*. I've included this section here as a placeholder, and a reminder of later coverage, as well as for completeness.

Summary

In this chapter, we zoomed in on component templates, and on what template syntax was available to us for creating them. Our template syntax included symbols, directives, and pipes.

We've seen that directives are just templateless components, and they come in two main flavors—**attribute directives**, and **structural directives**. Whatever their flavor or category, we associate (or attach) directives to HTML elements by adding them as attributes of those elements.

We've gone over the following special symbols that we can use in our template syntax. We've also covered the built-in directives that we can use in our template syntax. Next, we covered event binding, as well as property binding, and finally, we covered pipes, which provide us with the means to format our data so that it can be rendered to our views in the way we desire.

We know that Angular is extensible and that it provides us with mechanisms to create custom directives, as well as custom pipes—but we are deferring our discussion of anything custom to Chapter 14, *Advanced Angular Topics*.

In the next chapter, Chapter 8, *Working with NG Bootstrap*, we're going to put our component hats back on our heads so that we can explore what `ng-bootstrap` brings to the table for us in building our Angular applications.

8

Working with NG Bootstrap

Bootstrap is one of the most popular CSS frameworks around—and Angular is one of the most popular web application frameworks around. NG Bootstrap is a collection of widgets (that is, components) that are built from Bootstrap 4 CSS. They are tailor-made to be used as Angular components, and are intended to be a complete replacement of Bootstrap components, which are powered by JavaScript. A few examples of JavaScript-powered Bootstrap components include the following:

- Carousel
- Collapse
- Modal
- Popovers
- Tooltips

In this chapter, we're going to continue our exploration of components, but will focus on `ng-bootstrap`, which is a third-party Angular component library, as opposed to being a part of the Angular code base. This chapter, and [Chapter 9, Working with Angular Material](#), are relatively short chapters, but I wanted to include them in this book for the same reason I mentioned for having included [Chapter 5, Flex-Layout – Angular's Responsive Layout Engine](#)—and that is to give you choices. In the context of this chapter, that means having choices of ready-made components that you can leverage for your Angular applications.

`ng-bootstrap` doesn't have an official acronym, but for the purposes of convenience, in this chapter, I'm going to give it one. We'll be referring to NG Bootstrap as NGB—which, as it turns out, is also a fun thing to type on the keyboard (since the letters are so closely placed together). Try it.

As in all the other chapters in this book, I will not consume page upon page to simply regurgitate NGB's official documentation, which is freely available online, just to make the book seem formidable. I'd rather give you a 300-to 400-page book filled with hand-selected goodness that keeps you reading, rather than a 500-600-page book that can be used as a sleep narcotic, for your hard-earned money. That being said, NGB's official online documentation can be found here:

<https://ng-bootstrap.github.io>.

One last thing I wanted to mention really quickly is that this chapter and the following one ([Chapter 8, Working with NG Bootstrap](#)) will be a lot more visual than the other chapters in the book. This is because we're now getting into the actual meat of our example application, and we're going to start building things out visually.

With the housekeeping matters now out of the way, here's what we're going to be covering together in this chapter:

- Integrating NGB
- NGB widgets (specifically, collapse, modal, and carousel)
- Design rules of thumb that we should think about to help avoid overuse of widgets

Integrating NGB

NGB's purpose in life is to be a complete replacement for Bootstrap's components that require JavaScript (such as the components listed at the beginning of the chapter). In fact, on the very first page of the *Getting Started* section on their official website, they go further and say that you shouldn't be using any JavaScript-based components at all—or even their dependencies, such as jQuery or Popper.js. This can be found at: <https://ng-bootstrap.github.io/#/getting-started>.

Installing NBG

First things first: before we take a look at a caveat that we need to be aware of when using NGB, let's add it to our project—and I'll also show you how to fix the conflicting libraries that you may encounter (by showing you my `package.json` file).

Installing NGB is straightforward using `npm`. But, like other modules, we also need to import it and list it in our root module (that is, `app.module.ts`). Here are the steps:

1. Run `npm install: npm install --save @ng-bootstrap/ng-bootstrap`
2. Import NGB into our root module: `import {NgbModule} from '@ng-bootstrap/ng-bootstrap';`
3. List `NgbModule` in the imports array (as a parameter to the root module's `@NgModule` decorator) like this: `NgbModule.forRoot()`

If you create an Angular module that uses NGB, you'll have to import NGB into it as well. The syntax for importing NGB into other modules is the exact same as the one just outlined for importing it into our root module, but the syntax for listing the NGB module as a parameter to the module's `@NgModule` decorator is slightly different. It's just listed in the imports array as `NgbModule`, as opposed to `NgbModule.forRoot()`, as we have to list it in our root module.

So, how are we going to take a look at a few of their components without unintentionally messing up the NGB portions of our example application? There's only one way—we're going to make sure that we do not directly or indirectly load `jQuery` or `Popper.js` into our example application, by not using the Bootstrap components (make sure you understand that Bootstrap and NGB are two different libraries).

Let me clarify something real quick. We have `jQuery` and `Popper.js` installed in our example application, and you can verify this by taking a look at our `package.json` file. In it, you will see entries for `jQuery` and `Popper.js` listed in the dependencies section. We are not going to be uninstalling these libraries. They are harmless to our use of NGB as long as we don't load them by also using Bootstrap. To put it another way, NGB components and Bootstrap components should not coexist in our Angular applications. We can use one or the other without issue—but never both. Does that make sense? OK, good.

If you try to remove `jQuery` and/or `Popper.js` from the project, you will likely get several compile warnings whenever you run the project. While warnings may not prevent the project from running, always try to strive for clean builds.

The chore of ensuring you get a clean build can sometimes be a pain in the neck, because you need to pay attention to the versions of your libraries. The code listing that follows is my package.json file. I consistently get a clean install compilation when I run `npm install` and then `npm start`. If you're not getting a clean compilation, you may want to compare your package.json against mine, as shown here:

```
{  
  "name": "listcaro",  
  "version": "0.0.0",  
  "license": "MIT",  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve -o",  
    "build": "ng build --prod",  
    "test": "ng test",  
    "lint": "ng lint",  
    "e2e": "ng e2e"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/animations": "^6.0.4",  
    "@angular/cdk": "^6.2.1",  
    "@angular/common": "^6.0.4",  
    "@angular/compiler": "^6.0.4",  
    "@angular/core": "^6.0.4",  
    "@angular/flex-layout": "^6.0.0-beta.16",  
    "@angular/forms": "^6.0.4",  
    "@angular/http": "^6.0.4",  
    "@angular/platform-browser": "^6.0.4",  
    "@angular/platform-browser-dynamic": "^6.0.4",  
    "@angular/router": "^6.0.4",  
    "@angular/material": "^6.2.1",  
    "@ng-bootstrap/ng-bootstrap": "^2.1.0",  
    "bootstrap": "^4.0.0",  
    "core-js": "^2.4.1",  
    "jquery": "^3.3.1",  
    "npm": "^6.1.0",  
    "popper": "^1.0.1",  
    "popper.js": "^1.14.3",  
    "rxjs": "^6.0.0",  
    "save": "^2.3.2",  
    "zone.js": "^0.8.26"  
  },  
  "devDependencies": {  
    "typescript": "2.7.2",  
    "@angular/cli": "~1.7.4",  
    "@angular/compiler-cli": "^6.0.4",  
  }  
}
```

```
    "@angular/language-service": "^5.2.0",
    "@types/jasmine": "~2.8.3",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codelyzer": "^4.0.1",
    "jasmine-core": "~2.8.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~2.0.0",
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~4.1.0",
    "tslint": "~5.9.1"
  }
}
```



You can take a look at the list of available Angular modules, and their latest version numbers, that you can install with npm at: <https://www.npmjs.com/~angular>.

Why use NGB?

With the restrictions of not being able to have JavaScript-based components, nor directly using JavaScript libraries such as jQuery or Popper.js, you may be asking, *why use NGB at all?*

That's a good question. Here's the short answer, in point form:

- Angular does not depend on jQuery. It uses its own implementation of jQuery, called jQLite, which is a subset of jQuery.
- We don't lose out on being able to use any Bootstrap components that are powered by JavaScript (such as modal or carousel) because they are re-done for Angular in NGB. Again, NGB's sole purpose in life is to completely replace any JavaScript-powered Bootstrap components.

- A rule of thumb when building Angular applications is to try to only use Angular-specific components; that is to say, components that are specifically made for Angular—such as NGB widgets and components from Angular Material. This, of course, includes creating your own custom Angular components. Though you can work around this by jerry-rigging non-Angular-specific components, it's not recommended. Angular is full featured, and as we've learned, it's also extremely extensible. It would be very difficult to think of a use case where sticking to Angular-specific components, modules, directives, pipes, services, and so on would prevent you from doing what you needed to do.
- NGB is a solid Angular-centric component library and works well when you don't try to create workarounds that are discouraged.

Creating our playground for NGB (and Angular Material, and more)

There are only two dependencies for NGB (Angular and Bootstrap CSS), and luckily our example application already has these two things in place—one by default (since our example application is an Angular application), and the other from our having installed Bootstrap during Chapter 3, *Bootstrap – Grid Layout and Components*. However, we are going to add something to our example application so that we can experiment with using NGB components—a playground view.

A long-standing traditional thing I do when building web applications of any technology stack, and not just for Angular applications, is to add a page as a place where I can experiment with stuff within the context of the application that I'm building at the time. I refer to that as a playground. In our case, our playground will be a component whose template will act as our experimental canvas as we explore a few NGB components. We're also going to hook it up to our menu so that we can access it easily.

We'll hang on to our playground view throughout the rest of the book, only deleting it in Chapter 15, *Deploying Angular Applications*, where we'll learn how to deploy our application and won't want our playground to go along for the ride.

So, let's do that now. It has been a while since we've added components to our example application that we created in Chapter 4, *Routing*, and so I wanted to take this opportunity to enumerate the steps to do this (within their own sections that follow) using the addition of the playground as the example. Note that this is the manual way of adding a component to our project—unlike using the CLI to add it for us, as we did a few chapters ago.

Creating a playground directory

The first thing we need to do is to create a directory that will hold the files we'll need for our playground component. Each of our components has its own directory, and are all subdirectories the `app` directory—which is itself a subdirectory of the `src` directory within the project root directory.

Since we're adding a new component, we'll follow our convention and create a directory for it. In your IDE, right-click the `app` directory, select `New Folder`, and enter `playground` as the name—which follows our convention that we've used so far. Once that is done, we'll have a place to insert the files that will collectively make up our component.

Creating the playground component class

We now need to create our playground component class. In your IDE, right-click the newly created `playground` directory and select `New File` and enter `playground.component.ts` as the name. The `playground.component.ts` file our component class. Enter the following code in this file:

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

@Component({
  selector: 'playground',
  templateUrl: './playground.component.html',
  styleUrls: ['./playground.component.css']
})
export class PlaygroundComponent implements OnInit {

  constructor() { }

  ngOnInit() { }

  pageTitle: string = "Playground";
}
```

By looking at our `playground` Component class file, you'll notice a few things:

- We're importing `OnInit`, in addition to the component from the `@angular/core` module. This is because we're giving ourselves a place to set up some variables if we need to—such as for passing in any child components.

- We've included a constructor for our class. Whether we use it or not, it provides us with a mechanism to tap into our component's life cycle to trigger some code if we wanted to. We won't use this right now, but I wanted to show you that our components function as traditional object-oriented classes, and thus have a constructor that we can leverage.
- We've set up our component to use external files for its template and its style, as opposed to having them inline. So, the next piece of business is to create these two files (see the following two sections).
- We have a single property (that is, `pageTitle`) declared in our class as a string, and we've assigned the name of our view to it. Our template in the following section displays this property using the one-way binding syntax.

Creating the playground template file

We now need to create the template file for our playground component, which will be our component's visual interface. In your IDE, right-click the `playground` directory, select `New File`, and enter `playground.component.html`. The `playground.component.html` file is required because we've passed it into our component decorator as one of the parameters. Enter the following code in this file:

```
<h3>
  {{ pageTitle }}
</h3>
<hr>
```

There's not much in this file yet—but this is where we'll be adding the NGB components so that we can experiment with them. Experimenting, of course, is the very best way to learn any technology that may be new to you. The only thing our template does for now is to display our page name—by binding to our class's `pageTitle` property.

Creating the playground CSS file

The final file we need to create for our playground component is that which will house its styles. In your IDE, right-click the `playground` directory, select `New File`, and enter `playground.component.css` as the name. The `playground.component.css` file is also required because we've passed it into our component decorator as one of the parameters. Enter the following code in this file:

```
/* Nothing here yet. This is a placeholder file that we may use later. */
```

The preceding code is self-explanatory. We don't have any styles in this file yet—but it's good to create at least one CSS file for every component that you create.

Creating the playground menu item

OK. So, after from following the directions in the preceding sections, you should now have a playground component that you can use as a sandbox for experimenting with almost anything you want. In our immediate case, we'll be using it for experimenting with NGB widgets (that is, components), but we'll also be using this sandbox during Chapter 9, *Working with Angular Material*.

Before we move on to inserting the first NGB widget, we'll be taking a look at. It's a good idea to create a temporary menu link for our playground view so that we can get to it easily from within our application. Let's do this now.

In your IDE, open your `app.component.html` file. This is the main, or starting, a template that is loaded for your Angular application during the bootstrapping process. It is also where we created our menu in chapter 4, *Routing*. In this file, insert the following code after the `listings` menu item:

```
<li routerLinkActive="active" class="nav-item">
  <a routerLink="playground" class="nav-link">Playground</a>
</li>
```

All this small HTML code snippet does is to add a playground navigation link in our menu, and instruct Angular's routing system to load the playground component (and thus the playground template, and then recursively load any child components) when it is clicked.

OK, good—we're now all set up and ready to take a look at our first NGB widget.

NGB widgets

As previously mentioned, NGB widgets are third-party Angular components that are designed to replace the JavaScript-driven Bootstrap CSS components. NGB has many widgets available, but we're only going to be looking at three of them in the following sections.

You can find the entire list of NGB widgets, along with their documentation, at: <https://ng-bootstrap.github.io/#/components/>.

Collapse

The collapse component is a useful thing to use to preserve screen real estate. Our use case for the use of this component will be for toggling instructions to be either displayed or hidden. The state of the component would initially be collapsed when its parent component's template is rendered, but the user would be able to toggle the instructions to be shown and re-collapse them as needed.

Let's take a look at a quick example in the code that we can try out in our playground which toggles the display of a section of the page on and off—in our case, this content will be hypothetical instructions (for now).

We need to modify three files to make this work. The use of other NGB components (and even the Angular Material components that we'll be looking at in the next chapter) work similarly, so I'll take the time to explain things after each code listing because this is the very first third-party component that we're looking at together. When looking at similar components later on, any explanations will given if they differ substantially from these ones.

Our parent component

For this chapter, as well as Chapter 8, *Working with NG Bootstrap*, our parent component will always be our playground component.

Modify your playground component template (which is `playground.component.html` file) so that it looks as follows:

```
<h3>
  {{ pageTitle }}
</h3>
<hr>

<ngb-collapse></ngb-collapse>

<br />

This is our page's main content
```

The only new thing we've added that is of any importance to our playground template is `<ngb-collapse></ngb-collapse>`, which is our custom directive that will instruct Angular to insert our child component's template there. `ngb-collapse` is the selector in our component class's metadata (that is, the object we passed to the component decorator). Let's take a look at that file next.

Our NGB collapse component class

We've named our component class (which leverages NGB's collapse component) `NgbCollapseComponent`—but where does this code live? Well, we need to create a new directory and two new files within that directory just like we did when we created our playground component. Yes—we created three files for our playground component, but we'll be skipping the CSS file for `NgbCollapseComponent`.

First, create a directory called `ngb-collapse`. Within that new directory, create a file named `ngb-collapse.component.ts` and add the following code in it:

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

@Component({
  selector: 'ngb-collapse',
  templateUrl: './ngb-collapse.component.html'
})
export class NgbCollapseComponent {
  public isCollapsed = true;
}
```

As you can see, we've not defined a `styleUrls` array, which is why we don't require a file for it (which we would have named something like `ngb-collapse.component.css` if we wanted this component to have styling). For the purposes of experimenting with the NGB collapse component, we only care about creating a component class file and its template file.

The other thing of interest to us in our component class file is the `isCollapsed` property. We can, of course, name it whatever we want, but the important thing is that it is declared and is initially set to `true`. We're going to use this property by binding its value to the `ngbCollapse` attribute within our template file. Doing so will cause a part of our component template to be either collapsed (hidden) or expanded (displayed). Note that I emphasized that our targeted content within our component will either be hidden or displayed, as opposed to being either added to or removed from the DOM. If our content is hidden (that is, non-visible), it is still in the DOM. This is because the NGB collapse widget does not function as a structural directive. It achieves its hide/show functionality via attribute binding.

Let's now take a look at the third file, the component template for our `NgbCollapseComponent` class.

Our NGB collapse component template

Create another file within the `ngb-collapse` directory, name it `ngb-collapse.component.ts`, and add the following code in it:

```
<p>
  <button type="button" class="btn btn-outline-primary"
(click)="isCollapsed = !isCollapsed">
    {{ isCollapsed ? 'Show' : 'Hide' }} Instructions
  </button>
</p>
<div id="collapseExample" [ngbCollapse]="isCollapsed">
  <div class="card">
    <div class="card-body">
      These are the hypothetical instructions for something.
    </div>
  </div>
</div>
```

Let's look at this code together. The first thing of interest to us is the binding of the `click` event to the expression, which basically toggles our `isCollapsed` variable, defined in our component class, between `true` and `false`:

```
(click)="isCollapsed = !isCollapsed"
```

The text for our toggle button is always set to one of two values. When the instructions are displayed, the button text reads **Hide Instructions**. When the instructions are hidden, the button text reads **Show Instructions**. This is, of course, the behavior we want, but at first glance, you may assume that it takes an `if .. else` construct to make it all work.

Surprisingly, thanks to Angular's interpolation template syntax, it only takes a tiny amount of code to alter the button's text depending on the value of our `isCollapsed` variable.

Let's take a moment to see the tiny code snippet responsible for determining what the button text should be, and how it renders it for us:

```
{{ isCollapsed ? 'Show' : 'Hide' }} Instructions
```

In Chapter 7, *Templates, Directives, and Pipes*, we took a look at all the symbols that we can use in our template syntax—such as interpolation, two-way binding, and so on. The symbol that works its magic for us, in this case, is the interpolation symbol (that is, the set of double curly braces). The reason I call it magical is that not only does it serve as a string interpolation, but it is also smart enough to handle expressions and even function calls. So, we're not restricted to just having a variable name being treated as a simple string interpolation.

To determine what our button text should be, we use JavaScript's ternary operator syntax to render (or interpolate) the text to one of two values, **Show** or **Hide**, based on the value of our `isCollapsed` variable. Of course, whatever the Boolean value is, the *Instructions* text will always be rendered, resulting in the button text being either **Show Instructions**, or **Hide Instructions**. This is all done succinctly, and inline. Pretty cool, isn't it?

Importations and declarations

If you try and run the project at this point, you'll get a few errors. This is because we haven't set up our imports and declarations in our `app.module.ts` file for this component yet. Let's do this now.

Add this import line after the import line we added for our playground component:

```
import { NgbCollapseComponent } from './ngb-collapse/ngb-collapse.component';
```

And add `NgbCollapseComponent` to the declarations array.

With the preceding import and addition of our component class to the declarations array in the `app.module.ts` file, our project should build and run just fine.

Good job. Let's now move on to our modal component.

Modal

Modal dialog windows have been around since the early days of the Windows OS on desktops (pre-internet), and have become popular for websites as well—this is especially true since jQuery came onto the scene. Modal windows are used for interacting with the user—typically, to get information from them. Moreover, they help the designers focus the users' attention to where it should be by dimming the background for contrast, as well as disabling any interaction anywhere outside the modal area. One of our use cases for the use of a modal window will be for displaying the login form.

Let's take a look at a quick example in the code that we can try out in our playground, to display a modal window. Since the integration of NGB widgets all follow the same pattern, I won't cover it in as much detail as the collapse NGB widget, but I'll point the important areas.

Our components all start off the same way. We need to create a folder for our component (let's name it `ngb-modal`) and we need to create our two files—one for our component class, and the other for our component template. Let's name them `ngb-modal.component.ts` and `ngb-modal.component.html`, respectively.

In the sections that follow are the two code listings for our NGB modal component, followed by the necessary imports and declarations—just like we did for our collapse component.

Our NGB modal component class

In our component class, we first import the necessary classes from the appropriate modules, and we then decorate our class with the `@Component` decorator, so we can link it to a template and set up our selector (that is, our custom HTML tag that we'll add to our playground template).

Next, we add a constructor so we can inject the `NgbModal` service (note: we'll be covering dependency injection in Chapter 12, *Integrating Backend Data Services*).

Our class has a variable named `closeResult`, which is populated with a string (by the private method named `getDismissReason`), describing how the modal dialog is dismissed by the user.

We also have an `open` method that is responsible for causing the modal dialog to render. As we'll see in the code listing in the next section (on our component template), the `open` method is triggered by a button click from within our playground.

You'll notice that the `open` method takes a parameter (named `content` in this example). Our component's template wraps the content that is to be displayed in our modal dialog within its `ng-template` tags, and as you'll see, these tags have the `#content` template variable associated with them. If you remember from Chapter 7, *Templates, Directives, and Pipes*, the hash symbol (that is, `#`) in template syntax is used to denote a variable:

```
import {Component} from '@angular/core';
import {NgbModal, ModalDismissReasons} from '@ng-bootstrap/ng-bootstrap';

@Component({
  selector: 'ngb-test-modal',
  templateUrl: './ngb-modal.component.html'
})
export class NgbModalComponent {
  closeResult: string;
```

```
constructor(private modalService: NgbModal) {}

open(content) {
  this.modalService.open(content).result.then((result) => {
    this.closeResult = `Closed with: ${result}`;
  }, (reason) => {
    this.closeResult = `Dismissed ${this.getDismissReason(reason)}`;
  });
}

private getDismissReason(reason: any): string {
  if (reason === ModalDismissReasons.ESC) {
    return 'by pressing ESC';
  } else if (reason === ModalDismissReasons.BACKDROP_CLICK) {
    return 'by clicking on a backdrop';
  } else {
    return `with: ${reason}`;
  }
}
```

Let's now take a look at our component template, `ngb-modal.component.html`.

Our NGB modal component template

Our component template is not only responsible for supplying our view with the content to be displayed in the modal dialog, but will also supply us with the visual element that our users will use (in this case, a button) to trigger the modal dialog.

The following HTML code is our component template, which we will use later for our login form (note: we will be covering forms in Chapter 10, *Working with Forms*):

```
<ng-template #content let-c="close" let-d="dismiss">
  <div class="modal-header">
    <h4 class="modal-title">Log In</h4>
    <button type="button" class="close" aria-label="Close"
(click)="d('Cross click')">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
  <div class="modal-body">
    <form>
      <div class="form-group">
        <input id="username" class="form-control" placeholder="username" >
        <br>
        <input id="password" type="password" class="form-control" >
      </div>
    </form>
  </div>
</ng-template>
```

```
placeholder="password" >
    </div>
</form>
</div>
<div class="modal-footer">
    <button type="button" class="btn btn-outline-dark" (click)="c('Save
click')">submit</button>
    </div>
</ng-template>

<button class="btn btn-lg btn-outline-primary"
(click)="open(content)">Launch test modal</button>
```

Now that we have our component class and our component template, we have to tell our application's root module about them—and we'll do just that in the next section.

Importations and declarations

Just as with our collapse component, if you try and run the project at this point, you'll get a few errors—and for the same reasons—since we've not set up our imports and declarations in our `app.module.ts` file for this component. You know the drill.

Add this import line after the import line we had added for our playground and collapse components:

```
import { NgbModalComponent } from './ngb-modal/ngb-modal.component';
```

And add `NgbModalComponent` to the declarations array.

I know you're getting the hang of this. Let's get some more practice with this by integrating one more NGB widget to our playground view—and as a bonus, we'll take a sneak preview of Angular's `HttpClient` module. We'll be using the `HttpClient` module to fetch the images for our carousel, and we'll also be using the `HttpClient` module to call our APIs in Chapter 11, *Dependency Injection and Services*.

So let's stretch our legs and arms, fill up our cups with coffee, and move on to one of the more interesting components (and what will be the start of the show in our example application), the NGB carousel.

Carousel

The carousel component is most notably known as the tool (that is, the widget or component) to use for displaying a series of images in a predesignated order—much like flipping through a photo album. Our use case will be precisely that: giving the user the ability to flip through photos of the property.

Let's take a look at a quick example in the code that we can try out in our playground to display three images. We'll start with the component class, and then move on to the component template. These code listings are straight out of the carousel example on the NGB website, found at: <https://ng-bootstrap.github.io/#/components/carousel/examples>.

I leave the wiring up of the class, using the `import` statement and so on, to you as an exercise. Hint: it's exactly the same process that we previously covered when adding the collapse and modal components to our playground (in their respective *Importations and declarations* sections). However, I will make mention of a few things after each code listing.

Our NGB carousel component class

In this section, we will implement the `ngb-carousel` component class. The following is the updated component class. We will analyze the code in a bit:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';

@Component({
  selector: 'ngb-test-carousel',
  templateUrl: './ngb-carousel.component.html',
  styles: [
    '.carousel {
      width: 500px;
    }
  ]
})
export class NgbCarouselComponent implements OnInit {
  images: Array<string>;

  constructor(private _http: HttpClient) {}

  ngOnInit() {
    this._http.get('https://picsum.photos/list')
      .pipe(map((images: Array<{id: number}>) =>
```

```
this._randomImageUrls(images))
    .subscribe(images => this.images = images);
}

private _randomImageUrls(images: Array<{id: number}>): Array<string> {
    return [1, 2, 3].map(() => {
        const randomId = images[Math.floor(Math.random() *
images.length)].id;
        return `https://picsum.photos/900/500?image=${randomId}`;
    });
}
```

There are a few things going on in our component class, `ngb-carousel.component.ts`. We're importing the `HttpClient` class from Angular's `http` module, and we're also importing the `map` class from the `rxjs/operators` module. The `HttpClient` class, which we'll be looking at more closely in Chapter 11, *Dependency Injection and Services*, is used to fetch a JSON list of image objects from `https://picsum.photos`, a free service that serves up images as placeholders, providing, as their site says, **The Lorem Ipsum for photos**. The `map` class is used to randomly map three of the many image objects that are returned from the GET request of `HttpClient` to our string array variable, named `images`.

The fetching of the image objects from the API happens when our component is initialized because the GET request happens within the `ngOnInit()` component's life cycle hook.

Our NGB carousel component template

In this section, we will implement our `ngb-carousel` component template file:

```
<ngb-carousel *ngIf="images" class="carousel">
    <ng-template ngbSlide>
        <img [src]="images[0]" alt="Random first slide">
        <div class="carousel-caption">
            <h3>First slide label</h3>
            <p>Nulla vitae elit libero, a pharetra augue mollis interdum.</p>
        </div>
    </ng-template>
    <ng-template ngbSlide>
        <img [src]="images[1]" alt="Random second slide">
        <div class="carousel-caption">
            <h3>Second slide label</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        </div>
    </ng-template>
    <ng-template ngbSlide>
```

```
<img [src]="images[2]" alt="Random third slide">
<div class="carousel-caption">
  <h3>Third slide label</h3>
  <p>Praesent commodo cursus magna, vel scelerisque nisl
consectetur.</p>
</div>
</ng-template>
</ngb-carousel>
```

This template is straightforward. Everything about it is hardcoded, except for the `src` property of the `img` HTML elements. Using square brackets around the HTML `img src` attribute is an example of property binding (as we learned in [Chapter 7, Templates, Directives, and Pipes](#)). In this case, the number of images in the carousel was known to be three. In practice, and as we will do in our example application, the template would normally make use of the `*ngFor` structural directive to iterate through an array of items of variable lengths.

Having gone through a few examples of integrating NGB widgets into our playground, we can now implement them in our application for real.

Implementing NGB into our example application

In the preceding section, *NGBwidgets*, we covered few components that are available in NGB. Of course, you know by now why I would never cover any more than a small number of the available components—right? If you said, *yes Aki, I know why. If you covered all the components, you'd basically just be duplicating documentation that is readily available elsewhere*, you'd be correct! Covering 3 out of 16 is plenty—it's almost 19% (which is practically the same as duplicating one out of every five pages of documentation!).

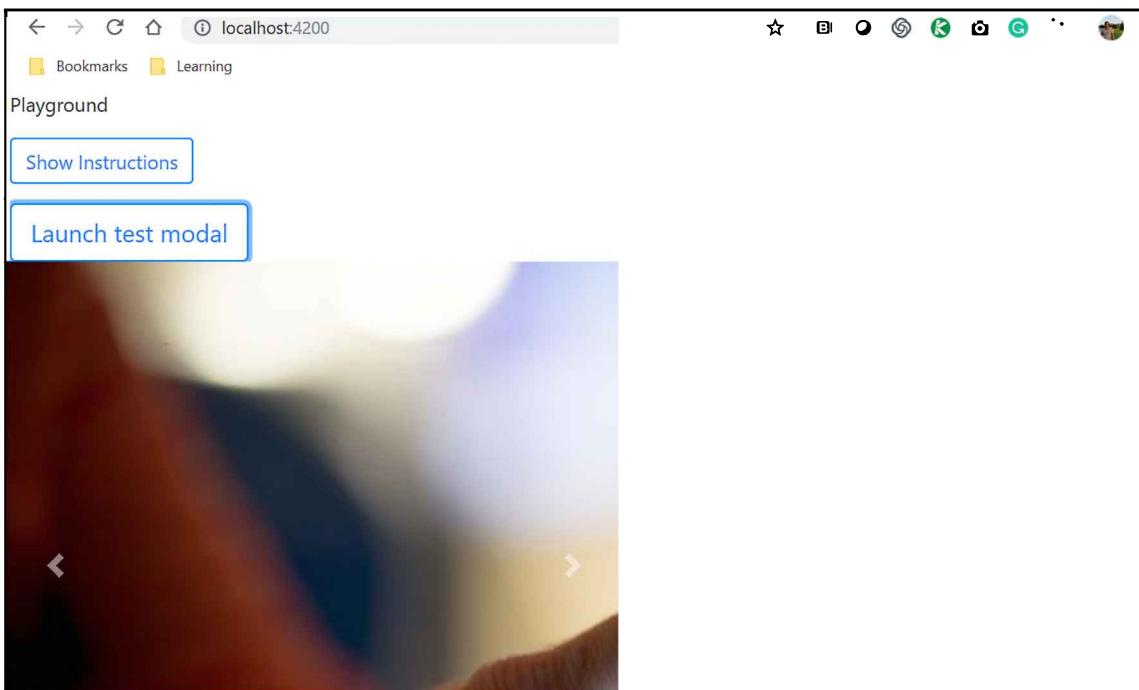
But there's also another reason. We're only going to implement two of the three NGB components that we covered—namely, the modal component and the carousel component—and so there is no need to cover too much more than those. OK, let's continue by putting our new found knowledge to practical use.

We learned about implementing the modal, carousel, and collapse components in earlier sections. We created selectors for each of the components. For the modal component, we created a selector named `ngb-test-modal`; for the carousel component, we created a selector named `ngb-test-carousel`; and last, but not least, for the collapse component, we created a selector named `ngb-collapse`. We now need to use these selectors in the `playground.component.html` file so that the widgets will be visible in the page.

The following is the updated code of the playground component template file:

```
<p>  
  {{pageTitle}}  
</p>  
  
<app-ngb-collapse></app-ngb-collapse>  
  
<app-ngb-modal></app-ngb-modal>  
  
<app-ngb-carousel></app-ngb-carousel>
```

We added the directives using the selectors for each of the components. Run the app using the `ng serve` command in the command line and we should see the output, as shown in the following screenshot:



Our application has the widgets integrated, but surely we can do a much better job with the design. In the next few sections, we will learn about some of the design principles and best practices that we will be implementing in the chapters to come.

UX design rules of thumb

There are rules of thumb for just about everything, and web design isn't any different. There are do's and don'ts in web design—and since we're now really starting to dive into our templates, it's a good time to review some of these design tenets.

There are probably several dozen design principals, but I'm not an expert on , **user experience (UX)**, and thus you'd be better served to pick up a good book that is focused on UX and GUI/interface design (I know that there are titles from Packt that you can look into). However, since we are building an application, and our application is made up of several components, It would be remiss if I didn't cover the fundamental three design principals.

Not only will we cover them in the following three short sections, but we will be adhering to them as we build our example application's templates. The reason we have things such as UX design principals essentially comes down to one thing—we want to have happy users!

Keep it clean

UX rule of thumb #1: Keep it clean.

Nothing gives users a headache faster than an overly busy (that is, cluttered) user interface. You may have heard of the expression *less is more*—and this expression certainly applies to UX design.

People feel like they have no time to do anything—and if doing something makes them feel like they are wasting their precious resource (that is, time), they become unhappy faster than you can count to 10. How does this relate to this first UX design principal? If there is a lot to look at on your page, they don't know where to start looking—and if they can't make sense of what they're looking at in short order, then you guessed it: they become unhappy.

Clutter is almost never a good thing. Think of your bedroom or kitchen. Are you happier when it's tidy and everything has a place and purpose, and you can easily and quickly find what you're looking for? Or are you happier when you waste 5 minutes looking for that spatula to cook your breakfast that you barely have time to eat? The answer, I hope, is obvious. Visitors to websites think the same way.

Keep it functional

UX rule of thumb #2: Keep it functional.

This UX rule of thumb is related to the first one, in that it is the same as saying that nearly everything on our view should have a function. The days of having a million bells and other objects on the screen that are nothing but eye candy are over. Do you remember the way websites looked in the late 1990s? Flash was all the rage. Web pages looked like snow globes, or had large animated buttons that pulsated, and read **Click Here Now**. These are no longer tolerated. Chances are excellent that if you have any such things on your web page, your visitor is going to leave your website as fast as they possibly can. If there is something on the screen, it had better have a purpose.

If you'd like to see an extreme example of a website that pays attention to the first and second (and the upcoming third) UX design principals, just take a look at Google's home page, at: <https://www.google.com/>.

Keep it obvious

UX rule of thumb #3: Keep it obvious.

Nothing frustrates users more than forcing them to use a large portion of their brain power, time, and detective skills just to find out what they need to do next, or how to do a specific task they'd like to perform in a web application.

Users of your web application are users for a reason, and that is that they need a tool to get something done. Whether the task that they want to get done is for pleasure or for work, it doesn't matter. Whatever it is they'd like to get done, they don't want to spend any more time than what is reasonable. If they need to spend too much time figuring things out, guess what? Yes! They become unhappy!

This third UX design principle is perhaps the most difficult one to adhere to, but it is our responsibility as application builders to give it the attention it deserves.

Summary

In this chapter, we explored NG Bootstrap—the first of the two third-party component libraries that are freely available to us for use in our Angular applications. We'll be exploring the second one, Angular Material, in the next chapter.

We took a look at how to install NGB, and then we created a playground within our application so we can have a place to play (that is, experiment) with these third-party components—including temporarily wiring the playground up to our menu via routing for easy navigational access to our playground. While we could have created a separate project altogether for playing around with components before integrating them into our application for their intended purposes, it is usually more convenient to create a playground within our existing infrastructure. Of course, when the comes for us to deploy our application, we can easily remove the playground and the menu option with its accompanying route.

With our playground all set up, we then dove in and took a look at how to integrate three of NGB's widgets: collapse, modal, and carousel.

Finally, to wrap the chapter up, since we're in the component and layout part of the book (as opposed to the backend data integration and services part of the book), it was a good time to cover a few design principle as well. So, we briefly covered three of the top tenants of good design: keeping it clean, functional, and obvious. We'll be adhering to these design principle as best we can throughout the remainder of the book.

Now then, keep your component hat firmly on your head, turn the page, and let's take a look at the gorgeously designed components that the Angular team cooked up for our use. Leveraging the Angular Material components in the right proportions and places will help boost our example application's usability and aesthetics. And fortunately, Angular Material plays nicely with Bootstrap, so there's no problem with having both libraries in the same Angular project.

9

Working with Angular Material

Welcome to the chapter on Angular Material. I must say, I'm impressed. Statistics show that most of the people who buy tech books don't get very far into them. You are a little more than halfway through the book—a job well done, Angular Jedi!

This will be a short chapter for a couple of reasons. First, this book is heavily intended to be used for building applications, primarily using Angular and Bootstrap. So think of this chapter as an added bonus for us. The other reason is this chapter is only intended to be an introduction to an alternative **user interface (UI)** component library, to Bootstrap when working with Angular. There should be a separate book on Angular Material alone, but this chapter will cover a lot of ground in terms of showing you the capabilities and components that the library offers.

We will learn about navigation and menu components, layout components, form field elements, buttons, dialog and pop-up components, and lots of fun elements that you will definitely appreciate, and possibly consider for the framework in your next project.

By summarizing, the topics we will be covering in this chapter are:

- What is Angular Material?
- Installing Angular Material
- Categories of components

Okay, let's get right to it by starting with the description of what Angular Material is.

What is Angular Material?

Angular Material is a rich collection of components, which can be easily plugged into Angular applications, and also works on web, mobile, and desktop applications. Material Design comes from Google, the makers of Angular, which essentially means that there is a lot of native support, optimization, and performance tuning done for the components, as well as for the new ones that will be rolled out in future. The following list shows some of the benefits we get when we consider using Material Design in our applications:

- The UI components are ready to be used right away, without any extra development efforts
- We can selectively choose to use the components individually, rather than being forced to import all the modules in one go
- The rendering of the components is extremely fast
- It is easy to plug data into components via a two-way or one-way data binding functionality, which is an extremely powerful feature of Angular
- The components have the same look, feel, and behavior across web, mobile, and desktop applications, which solves a lot of cross-browser and cross-device issues
- The performance is tuned and optimized for integration with Angular applications



You can find all the required documentation about Angular Material on the official website at: <http://material.angular.com>.

Before we proceed any further in this chapter, let's quickly generate the application where we will implement all of the Angular Material components. Run the following `ng` command in order to generate a new application named `AngularMaterial`:

```
ng new AngularMaterial
```

Once the command has been successfully executed, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter9_material>ng new AngularMaterial
? Would you like to add Angular routing? yes
? Which stylesheet format would you like to use? Sass [ http://sass-lang.com ]
CREATE AngularMaterial/angular.json (3976 bytes)
CREATE AngularMaterial/package.json (1315 bytes)
CREATE AngularMaterial/README.md (1032 bytes)
CREATE AngularMaterial/tsconfig.json (435 bytes)
CREATE AngularMaterial/tslint.json (1621 bytes)
CREATE AngularMaterial/.editorconfig (246 bytes)
CREATE AngularMaterial/.gitignore (629 bytes)
CREATE AngularMaterial/src/favicon.ico (5430 bytes)
CREATE AngularMaterial/src/index.html (302 bytes)
CREATE AngularMaterial/src/main.ts (372 bytes)
CREATE AngularMaterial/src/polyfills.ts (2841 bytes)
CREATE AngularMaterial/src/styles.scss (80 bytes)
CREATE AngularMaterial/src/test.ts (642 bytes)
CREATE AngularMaterial/src/browserslist (388 bytes)
CREATE AngularMaterial/src/karma.conf.js (1028 bytes)
CREATE AngularMaterial/src/tsconfig.app.json (166 bytes)
CREATE AngularMaterial/src/tsconfig.spec.json (256 bytes)
CREATE AngularMaterial/src/tslint.json (314 bytes)
CREATE AngularMaterial/src/assets/.gitkeep (0 bytes)
CREATE AngularMaterial/src/environments/environment.prod.ts (51 bytes)
CREATE AngularMaterial/src/environments/environment.ts (662 bytes)
CREATE AngularMaterial/src/app/app-routing.module.ts (245 bytes)
CREATE AngularMaterial/src/app/app.module.ts (393 bytes)
CREATE AngularMaterial/src/app/app.component.html (1152 bytes)
CREATE AngularMaterial/src/app/app.component.spec.ts (1122 bytes)
CREATE AngularMaterial/src/app/app.component.ts (220 bytes)
```

Now that our application has been generated, let's learn how to install the Angular Material library in our project.

Installing Angular Material

By now, you will have a strong gut feeling that when we want to install anything in Angular applications, we have a powerful **command-line interface (CLI)** tool. We will continue to use the same CLI, and with the help of npm, we will install Angular Material.

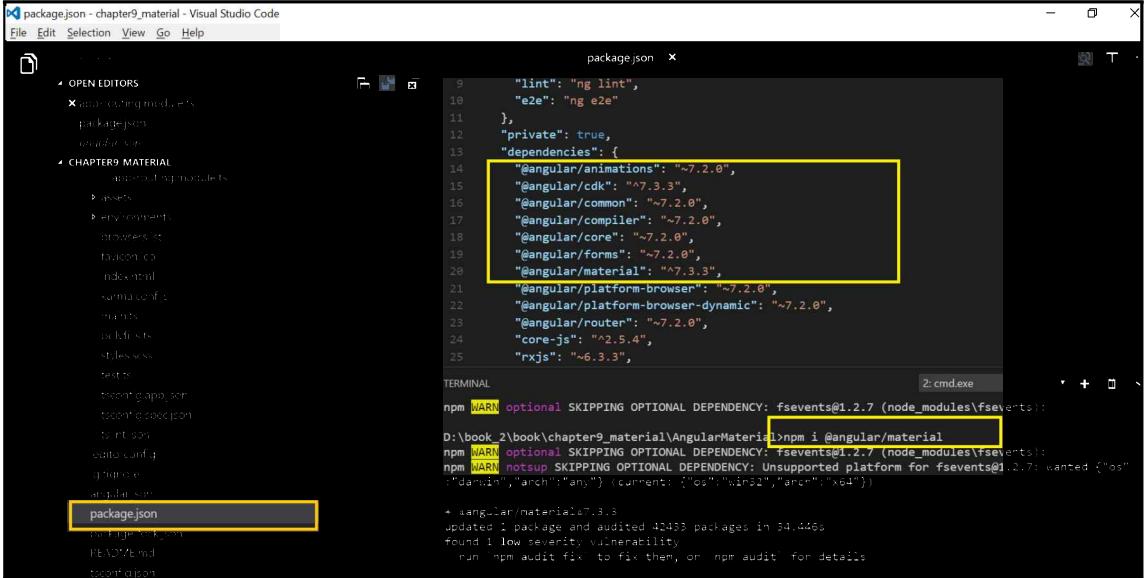


You can also choose to install Angular Material via the YARN command—different packaging systems, same outcome.

Angular Material has a core dependency and prerequisite to install two packages—CDK and Animations. So, let's install these first, and then we will install Angular Material:

```
npm i @angular/cdk --save  
npm i @angular/animations --save  
npm i @angular/material --save
```

After successfully running the preceding commands, we should see the output shown in the following screenshot:



The screenshot shows a Visual Studio Code interface with the package.json file open. The file contains the following JSON code:

```
9   "lint": "ng lint",  
10  "e2e": "ng e2e"  
11 },  
12 "private": true,  
13 "dependencies": {  
14   "@angular/animations": "~7.2.0",  
15   "@angular/cdk": "~7.3.3",  
16   "@angular/common": "~7.2.0",  
17   "@angular/compiler": "~7.2.0",  
18   "@angular/core": "~7.2.0",  
19   "@angular/forms": "~7.2.0",  
20   "@angular/material": "~7.3.3",  
21   "@angular/platform-browser": "~7.2.0",  
22   "@angular/platform-browser-dynamic": "~7.2.0",  
23   "@angular/router": "~7.2.0",  
24   "core-js": "~2.5.4",  
25   "rxjs": "~6.3.3",  
26 }  
27  
28 TERMINAL  
29 2: cmd.exe  
30 npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\fsevents):  
31 D:\book_2\book\chapter9_material\AngularMaterial>npm i @angular/material  
32 npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\fsevents):  
33 npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.7: wanted {"os":  
34 ("darwin","arch":"any")} (current: {"os":"win32","arch":"x64"})  
35 + angular-material@7.3.3  
36 updated 1 package and audited 42453 packages in 34.446s  
37 found 1 low severity vulnerability  
38   run `npm audit fix` to fix them, or `npm audit` for details
```

The dependencies section of the package.json file is highlighted with a yellow box. The package.json file itself is also highlighted with a yellow box at the bottom left.

Open the package.json file; we should see the packages that have been installed, and the corresponding version numbers listed alongside them. If you see the three packages that we have recently installed, it means we are ready to start creating some awesome UI screens using Angular Material.

Once we have installed Angular Material, we will need to import all the required modules into our `app.module.ts` file. There are a lot of modules provided by Material, each for a specific purpose. For example, we will need to import `MatCardModule` if we plan to use Material cards. Similarly, we need to import `MatChipsModule` if we want to use Material chips in our application. While we can definitely import only the required modules into `AppModule`, in most applications using Material UI, we will need all the modules. Now, let's quickly learn how to import all the modules in one go. We can import all the modules into a generic module, and then use the newly created generic module in the `app.module.ts` file. First, let's create a file in our project structure and name it `material-module.ts`, then we can add the following code to it in order to import all the modules into this file:

```
import {A11yModule} from '@angular/cdk/a11y';
import {DragDropModule} from '@angular/cdk/drag-drop';
import {ScrollingModule} from '@angular/cdk/scrolling';
import {CdkStepperModule} from '@angular/cdk/stepper';
import {CdkTableModule} from '@angular/cdk/table';
import {CdkTreeModule} from '@angular/cdk/tree';
import {NgModule} from '@angular/core';
import {
  MatAutocompleteModule,
  MatBadgeModule,
  MatBottomSheetModule,
  MatButtonModule,
  MatButtonToggleModule,
  MatCardModule,
  MatCheckboxModule,
  MatChipsModule,
  MatDatepickerModule,
  MatDialogModule,
  MatDividerModule,
  MatExpansionModule,
  MatGridListModule,
  MatIconModule,
  MatInputModule,
  MatListModule,
  MatMenuModule,
  MatNativeDateModule,
  MatPaginatorModule,
  MatProgressBarModule,
  MatProgressSpinnerModule,
  MatRadioModule,
  MatRippleModule,
  MatSelectModule,
  MatSidenavModule,
  MatSliderModule,
```

```
MatSlideToggleModule,  
MatSnackBarModule,  
MatSortModule,  
MatStepperModule,  
MatTableModule,  
MatTabsModule,  
MatToolbarModule,  
MatTooltipModule,  
MatTreeModule,  
} from '@angular/material';  
  
@NgModule({  
  exports: [  
    A11yModule,  
    CdkStepperModule,  
    CdkTableModule,  
    CdkTreeModule,  
    DragDropModule,  
    MatAutocompleteModule,  
    MatBadgeModule,  
    MatBottomSheetModule,  
    MatButtonModule,  
    MatButtonToggleModule,  
    MatCardModule,  
    MatCheckboxModule,  
    MatChipsModule,  
    MatStepperModule,  
    MatDatepickerModule,  
    MatDialogModule,  
    MatDividerModule,  
    MatExpansionModule,  
    MatGridListModule,  
    MatIconModule,  
    MatInputModule,  
    MatListModule,  
    MatMenuModule,  
    MatNativeDateModule,  
    MatPaginatorModule,  
    MatProgressBarModule,  
    MatProgressSpinnerModule,  
    MatRadioModule,  
    MatRippleModule,  
    MatSelectModule,  
    MatSidenavModule,  
    MatSliderModule,  
    MatSlideToggleModule,  
    MatSnackBarModule,  
    MatSortModule,
```

```
MatTableModule,  
MatTabsModule,  
MatToolbarModule,  
MatTooltipModule,  
MatTreeModule,  
ScrollingModule,  
]  
})  
export class MaterialModule {}
```

In the preceding code, we imported all the required modules into the file. Don't worry about categorizing the previously listed modules just yet. We learn about the modules when we learn about the components that are provided by Material. The next step is pretty obvious—we will need to import this newly created module into our `app.module.ts` file:

```
import {MaterialModule} from './material-module';
```

Once we have imported the module, don't forget to add it to the imports of `AppModule`. That's it. We are all set to start learning and implementing the components that are provided by Angular Material.



Did you know? Google has also released a lightweight CSS- and JavaScript-based, Lite library, Material Design Lite, which starts by using the components in the same way as in any other UI library. However, there may be some components that do not have full support. Learn more about it at <https://getmdl.io/>.

Let's jump right into learning about the components of Angular Material.

Categories of components

As a frontend developer, you will have used a lot of UI components, or even better, you might have created your own custom components in past projects. As previously mentioned, Angular Material provides a lot of components that can be readily and easily used in our applications. The UI components provided by Angular Material can be categorized under the following categories:

- Layouts
- Material cards
- Form controls
- Navigation

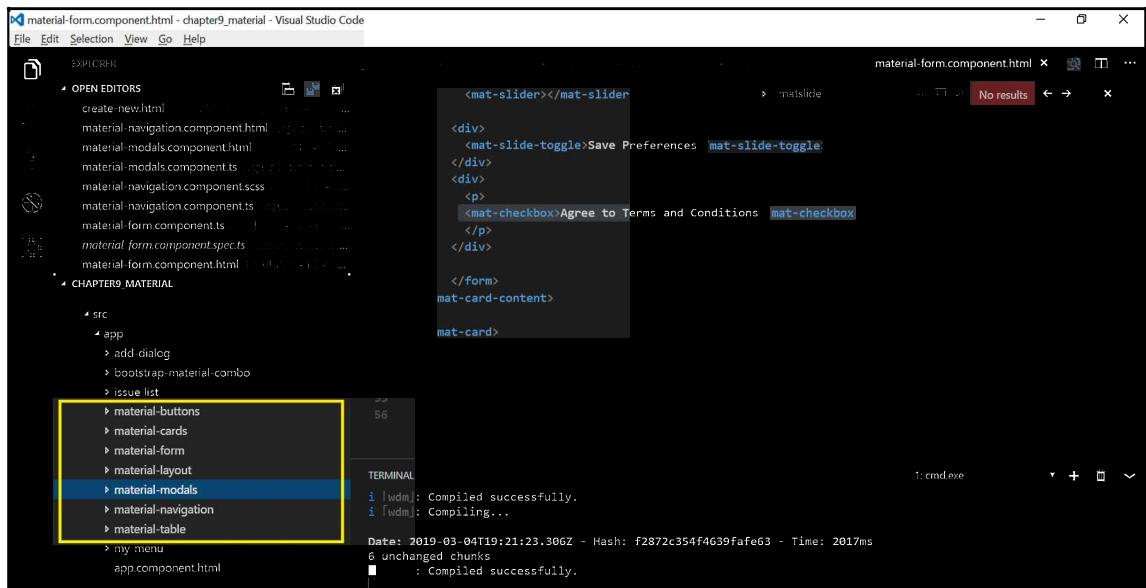
- Buttons and indicators
- Modals and popups
- Tables

It's a good idea to generate components for each of these categories, so that the placeholders will be available when we start implementing our application. These components will host all the components in a clearly categorized manner, and they will become your one-stop components that you can use to refer to any of the component implementations in the Material library.

First, let's generate the components for our categories. Run the following `ng` commands one after another:

```
ng g component MaterialLayouts
ng g component MaterialCards
ng g component MaterialForm
ng g component MaterialNavigation
ng g component MaterialButtons
ng g component MaterialModals
ng g component MaterialTable
```

Following the successful running of the commands, we should see that the components are generated and added to our project structure, as shown in the following screenshot:



Great. We have generated our application; we have installed Angular Material. We have also imported all the required modules into our `AppModule` file, and finally, we have generated the components for each category in the UI components of Material. The last thing we need to do before we start implementing the Material components is to add the routes for each of the previously listed categories. Open the `app-routing.module.ts` file, import all the newly created components, and add the routes to the file:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { MaterialFormComponent } from './material-form/material-
form.component';
import { MaterialNavigationComponent } from './material-
navigation/material-navigation.component';
import { MaterialCardsComponent } from './material-cards/material-
cards.component';
import { MaterialLayoutComponent } from './material-layout/material-
layout.component';
import { MaterialTableComponent } from './material-table/material-
table.component';
import { MaterialModalsComponent } from './material-modals/material-
modals.component';
import { MaterialButtonsComponent } from './material-buttons/material-
buttons.component';

const routes: Routes = [
  { path: 'material-forms', component: MaterialFormComponent },
  { path: 'material-tables', component: MaterialTableComponent },
  { path: 'material-cards', component: MaterialCardsComponent },
  { path: 'material-layouts', component: MaterialLayoutComponent },
  { path: 'material-modals', component: MaterialModalsComponent },
  { path: 'material-buttons', component: MaterialButtonsComponent },
  { path: 'material-navigation', component: MaterialNavigationComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

In the preceding code, we imported all the newly created components and created route paths for each of the them. So far, so good. Now, the big stage is all set and ready to be rocked. Let's start with our layouts first.

Navigation

One of the most common and basic necessities of any web application is a navigational menu or toolbar. Angular Material provides us with multiple options, with which we can choose the type of menu that is most suitable for our application.

Navigation components using schematics

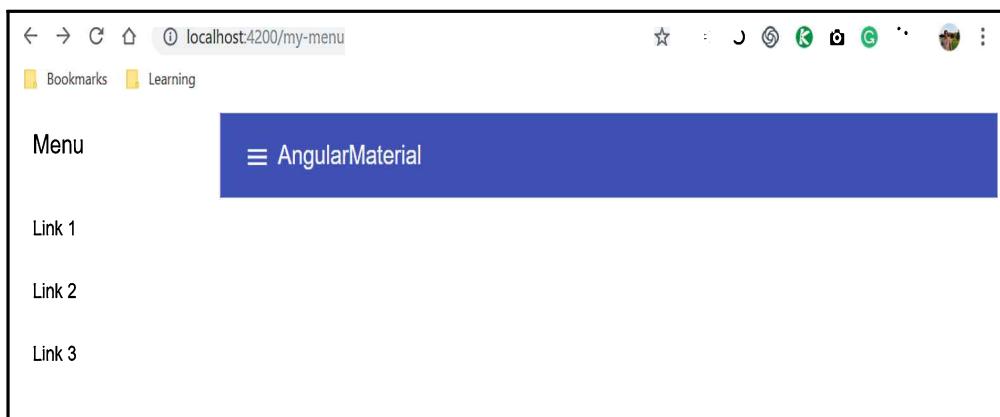
We will start with the simplest and fastest way to get the navigation added to our apps by using schematics. That's right, we are just a command away from getting our menu up and running. Angular CLI provides schematics in order to get a wide range of components. To install the navigation menu in our app, run the following command in the Angular CLI command prompt:

```
ng generate @angular/material:nav myMenu
```

In the preceding command, we used schematics to generate a new menu component called myMenu. Upon the successful running of the command, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter9_material\AngularMaterial>ng generate @angular/material:nav myMenu
CREATE src/app/my-menu/my-menu.component.html (936 bytes)
CREATE src/app/my-menu/my-menu.component.spec.ts (1083 bytes)
CREATE src/app/my-menu/my-menu.component.ts (559 bytes)
CREATE src/app/my-menu/my-menu.component.css (193 bytes)
UPDATE src/app/app.module.ts (3194 bytes)
```

Run the app using the `ng serve` command, and we should see the output shown in the following screenshot:



Isn't that a really cool navigation menu? It comes with a top header toolbar, and a sidebar menu that is collapsible . This component is autogenerated by the schematics. If you are not a big fan of autogenerated components, it's okay, we developers can be picky about these things. Let's see how we can create our own menu.

Custom Material menus and navigation

Angular Material provides a `MatMenuModule` module, which provides directives, `<mat-menu>`, and `MatToolBarModule`. Also provided is `<mat-toolbar>`, which will be used to implement the menu and the header in our application. Open the `material-navigation.component.html` file and add the following code:

```
<mat-toolbar id="appToolbar" color="primary">
<h1 class="component-title">
<a class="title-link">Angular Material</a>
</h1>
<span class="toolbar-filler"></span>
<a href="#">Login</a>
<a href="#">Logout</a>
</mat-toolbar>
```

In the preceding code, we implemented the toolbar directive using `<mat-toolbar>` as a wrapper, and we added a heading title using `<h1>`. We also added some links to the header sections. Run the app using `ng serve`, and we should see the output shown in the following screenshot:



That's wonderful. Let's enhance it a little more. We want to add a drop-down menu our header toolbar. Remember I told you that we have the `<mat-menu>` directive provided by the `MatMenuModule` module? Let's add the menu directive to the header toolbar in the preceding code as follows:

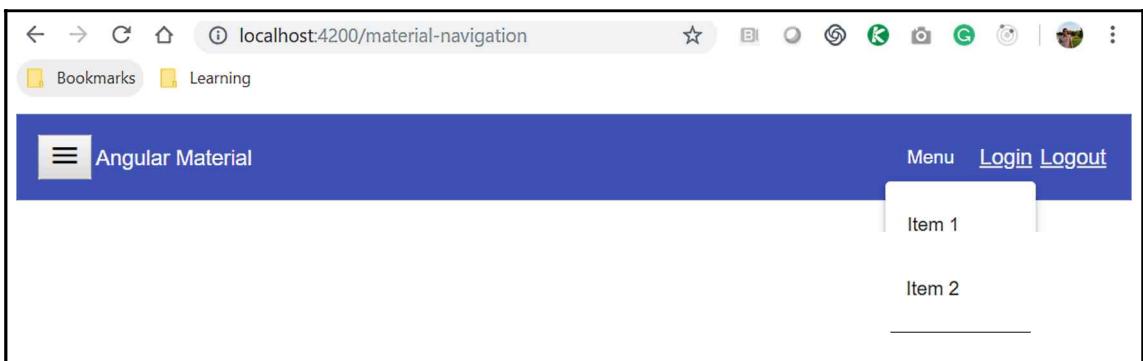
```
<mat-toolbar id="appToolbar" color="primary">
<button md-icon-button (click)="sidenav.toggle()" class="md-icon-button sidenav-toggle-button" [hidden]="sidenav.opened">
<mat-icon aria-label="Menu" class="material-icons">menu</mat-icon>
</button>

<h1 class="component-title">
<a class="title-link">Angular Material</a>
</h1>
<span class="toolbar-filler"></span>

<button mat-button [matMenuTriggerFor]="menu"
color="secondary">Menu</button>
<mat-menu #menu="matMenu" >
<button mat-menu-item>Item 1</button>
<button mat-menu-item>Item 2</button>
</mat-menu>

<a href="#">Login</a>
<a href="#">Logout</a>
</mat-toolbar>
```

Notice that we have added a button using the `mat-button` attribute, and we are binding the `matMenuTriggerFor` attribute. This will show the drop-down menu defined with `<mat-menu>` directives. Now let's run the app using the `ng serve` command, and we should see the output as follows:

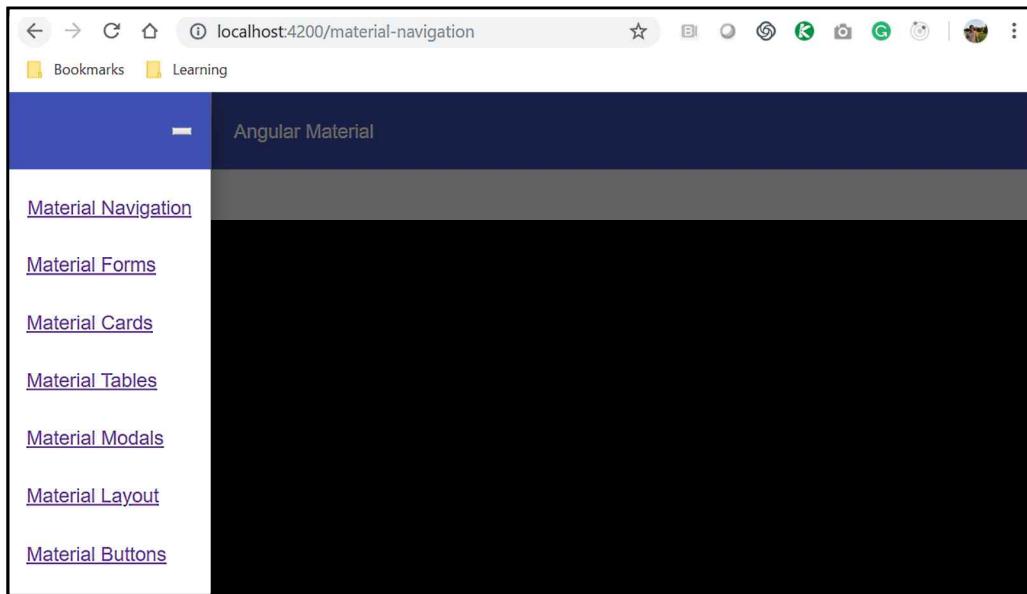


Custom sidebar menus

Awesome. So, now we have our custom-made menu ready to use. I know you want more, don't you? You want to add a sidebar too? Let's do it. To add the sidebar to our application, Angular Material provides us with a `MatSidenavModule` module, which provides the `<mat-sidenav>` directive that we can use in our application. So, let's continue to modify the preceding code as follows:

```
<mat-sidenav-container fullscreen>
  <mat-sidenav #sidenav mode="push" class="app-sidenav">
    <mat-toolbar color="primary">
      <span class="toolbar-filler"></span>
      <button md-icon-button (click)="sidenav.toggle()" class="md-icon-button
        sidenav-toggle-button" [hidden]="!sidenav.opened">
        </button>
    </mat-toolbar>
  </mat-sidenav>
  <mat-toolbar id="appToolbar" color="primary">
    <button md-icon-button (click)="sidenav.toggle()" class="md-icon-button
      sidenav-toggle-button" [hidden]="sidenav.opened">
      <mat-icon aria-label="Menu" class="material-icons">menu</mat-icon>
    </button>
    <h1 class="component-title">
      <a class="title-link">Angular Material</a>
    </h1>
    <span class="toolbar-filler"></span>
    <button mat-button [matMenuTriggerFor]="menu"
      color="secondary">Menu</button>
    <mat-menu #menu="matMenu" >
      <button mat-menu-item>Item 1</button>
      <button mat-menu-item>Item 2</button>
    </mat-menu>
    <a href="#">Login</a>
    <a href="#">Logout</a>
  </mat-toolbar>
</mat-sidenav-container>
```

Don't get scared by looking at the number of lines of code. We have just made a few changes, such as adding the `<mat-sidenav>` directive, which will contain the content of the sidebar. Finally, we are wrapping the entire content inside the `<mat-sidenav-container>` directive; this is important, as the sidebar will overlay on the content. Run the app using the `ng serve` command, and we should see the output shown in the following screenshot:



If you see the output shown in the preceding screenshot, give yourself a pat on the back. Kudos! You are doing absolutely wonderfully. So, we have learned two ways to implement the navigation and the menu in our applications. We can either use schematics to generate the navigation component, or we can write a custom menu navigation component. Either way, **user experience (UX)** wins!

Now that we have our navigational menu component, let's learn about the other components of the Angular Material library.

Cards and layout

In this section, we will learn about Angular Material cards and layouts. The basic layout component of Angular Material is a card. The card wrapper layout component can also include lists, accordions or expansion panels, tabs, steppers, and so on.

Material cards

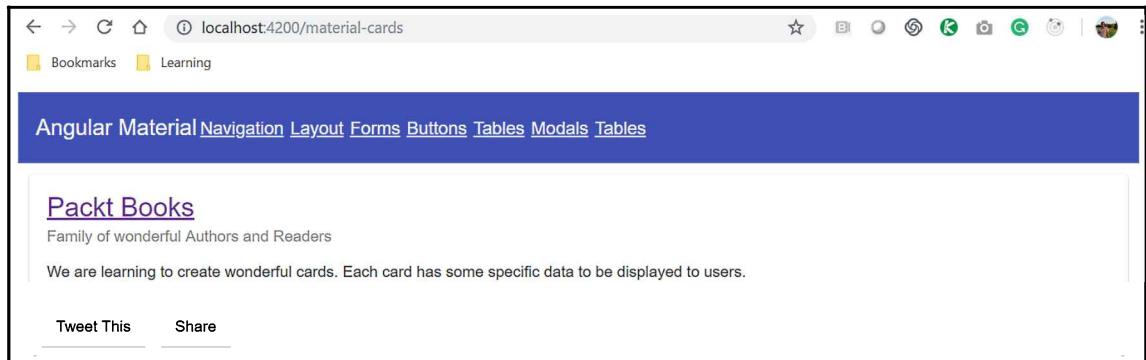
A card is a content container for text, images, links, and actions that are used to group the data of a single subject. Cards can have a header, a body, an image, or links, which can be displayed to the user based on their availability and functionality. Angular Material provides a module named `MatCardModule`, which provides the `<mat-card>` directive. We will use this to group the content of our application.

A basic example of creating a card is given as follows:

```
<mat-card class="z-depth" >
  <mat-card-title><a href="" primary>Packt Books</a></mat-card-title>
  <mat-card-subtitle>Family of wonderful Authors and Readers
  </mat-card-subtitle>
  <mat-card-content>
    We are learning to create wonderful cards. Each card has some specific
    data to be displayed to users.
  </mat-card-content>
  <mat-card-actions> <button mat-raised-button>Tweet This</button>
    <button mat-raised-button>Share</button></mat-card-actions>
</mat-card>
```

In the preceding code, we made use of the directives provided by `MatCardModule`. We will use `<mat-card>` as a wrapper directive in order to group the content. By using the `<mat-card-title>` directive, we are setting the title of the card. We are setting a subtitle by using the `<mat-card-subtitle>` directive inside the `<mat-card>` directive. Inside `<mat-card-content>`, we place all the content that we need to display to the user. Each card may have the actions that we want the user to perform, such as sharing, editing, approving, and so on. We can display the card actions using the `<mat-card-actions>` directive.

Run the app using the `ng serve` command, and we should see the output shown in the following screenshot:



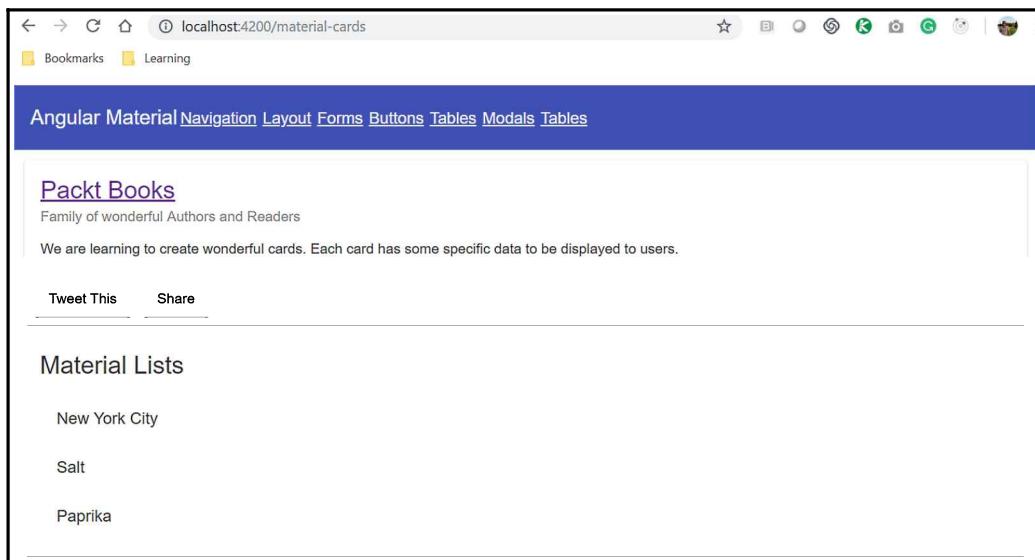
Notice that we have added some content inside the Angular Material card. Are you wondering what kind of content can be displayed inside the cards? You name it, and we can use it. We can add links, images, lists, accordions, steppers, and more. In the next section, we will learn how to add lists to our cards.

Lists

Lists are collections of items grouped together. We can have either ordered list, or unordered in our applications. In this section, we will learn how to add different types of lists inside the cards. Take a look at the following sample code:

```
<mat-card class="z-depth" >
  <mat-card-title>Material Lists</mat-card-title>
  <mat-card-content>
    <mat-list>
      <mat-list-item> New York City</mat-list-item>
      <mat-list-item> London</mat-list-item>
      <mat-list-item> Dallas</mat-list-item>
    </mat-list>
  </mat-card-content>
</mat-card>
```

In the preceding code, we added a list of a few cities. We used the `<mat-list>` and `<mat-list-item>` directives provided inside `MatListModule`, in order to create and display the list of cities inside the cards. The output of the preceding code is as follows:



Lists with dividers

We can also easily add a divider class for the list items in order to visually separate them into rows. We need to add the `<mat-divider>` directive in order to achieve that functionality. Take a look at the updated code as follows:

```
<mat-card class="z-depth" >
  <mat-card-title>Material Lists with Divider</mat-card-title>
  <mat-card-content>
    <mat-list>
      <mat-list-item> Home </mat-list-item>
      <mat-divider></mat-divider>
      <mat-list-item> About </mat-list-item>
      <mat-divider></mat-divider>
      <mat-list-item> Contact </mat-list-item>
      <mat-divider></mat-divider>
    </mat-list>
  </mat-card-content>
</mat-card>
```

Navigation lists

We can extend the lists to make them clickable, and therefore make them into navigational links. To make the list items clickable, we will need to use the `<mat-nav-list>` directive. Take a look at the sample code as follows:

```
<mat-card class="z-depth" >
  <mat-card-title>Material Navigational Lists</mat-card-title>
  <mat-card-content>
    <mat-nav-list>
      <a mat-list-item href="#" *ngFor="let nav of menuLinks"> {{ nav }} </a>
    </mat-nav-list>
  </mat-card-content>
</mat-card>
```

In the preceding code, we created a navigation type of list and the list items inside our cards, using the `<mat-nav-list>` and `<mat-list-item>` directives that are provided in the `MatListModule` module. The output of the preceding code is given as follows:



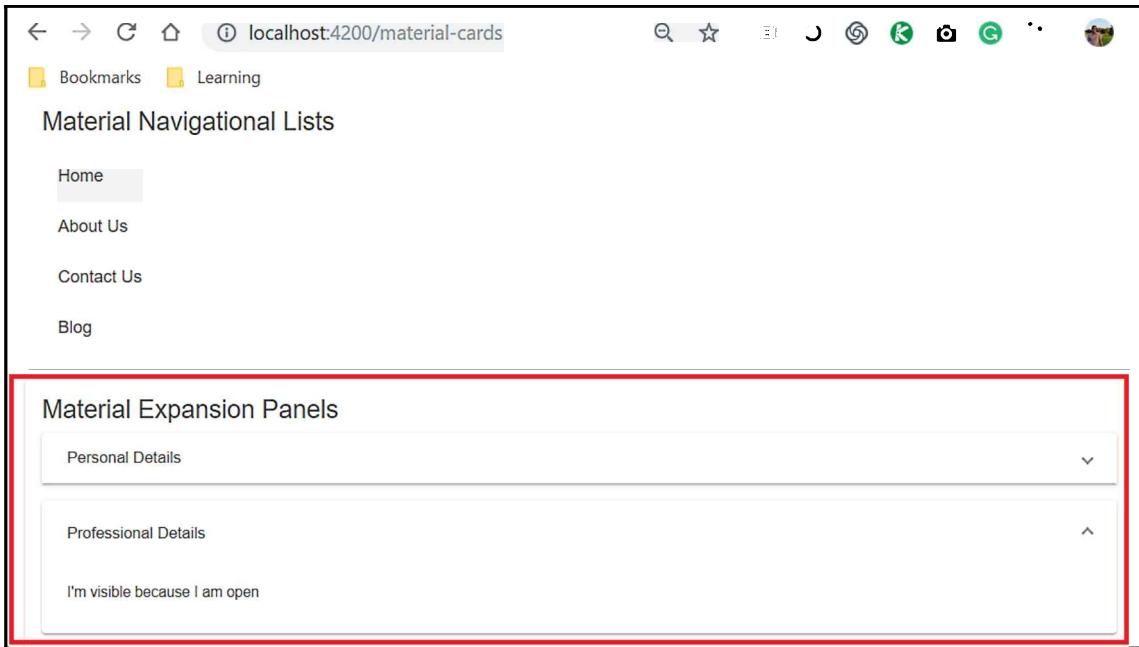
Accordions and expansion panels

One other very cool UI component is accordions, or expansion panels. It's very handy to use when we need to group data together. We will need to use `<mat-accordion>` and `<mat-expansion-panel>`, which are provided in the `MatExpansionModule` module, to implement the accordion functionality in our application. Take a look at the following sample code:

```
<mat-card class="z-depth" >
  <mat-card-title>Material Expansion Panels</mat-card-title>
  <mat-card-content>
    <mat-accordion>
      <mat-expansion-panel>
        <mat-expansion-panel-header>
          <mat-panel-title>
            Personal Details
          </mat-panel-title>
        </mat-expansion-panel-header>
      </mat-expansion-panel>
      <mat-expansion-panel>
        <mat-expansion-panel-header>
          <mat-panel-title>
            Professional Details
          </mat-panel-title>
        <mat-panel-description>
          </mat-panel-description>
        </mat-expansion-panel-header>
        <p>I'm visible because I am open</p>
      </mat-expansion-panel>
    </mat-accordion>
  </mat-card-content>
</mat-card>
```

```
</mat-expansion-panel>
</mat-accordion>
</mat-card-content>
</mat-card>
```

Each `<mat-expansion-panel>` will have a `<mat-expansion-panel-header>`, where we can provide the title and description for the expansion panel, and we place the content inside the `<mat-expansion-panel>` directive itself. The output of the preceding code is as follows:



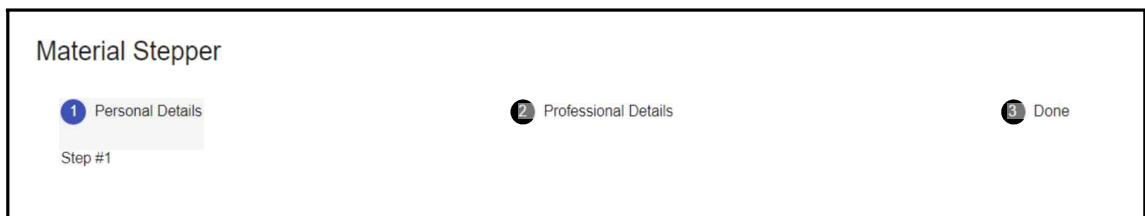
There will be use cases where we need to walk the user through a series of steps. That's where our next component comes into the picture. It's called a stepper. As the name suggests, this will be used to design steps either horizontally or vertically, and will group a series of steps that the user can navigate to.

Steppers

Similar to what we learned in the *Accordions and expansion panels* section, we will need to add a wrapper, and a `<mat-horizontal-stepper>` directive, and inside that, we will create `<mat-step>` directives. For each step that we want to add, we will need to create a new `<mat-step>` directive for our application. We can also create a vertical stepper. For that, the wrapper class we will use is the `<mat-vertical-stepper>` directive. Take a look at the following code; we are creating a horizontal stepper:

```
<mat-card class="z-depth" >
<mat-card-title>Material Stepper</mat-card-title>
<mat-card-content>
<mat-horizontal-stepper [linear]="isLinear" #stepper>
<mat-step label="Personal Details">
Step #1
</mat-step>
<mat-step label="Professional Details">
Step #2
</mat-step>
<mat-step>
<ng-template matStepLabel>Done</ng-template>
You are now done.
<div>
<button mat-button matStepperPrevious>Back</button>
<button mat-button (click)="stepper.reset()">Reset</button>
</div>
</mat-step>
</mat-horizontal-stepper>
</mat-card-content>
</mat-card>
```

In the preceding code, we created a horizontal stepper with three steps. To define the stepper, we have used `<mat-horizontal-stepper>`, and for defining the actual steps, we have used the `<mat-step>` directive. The output of the preceding code is given as follows:

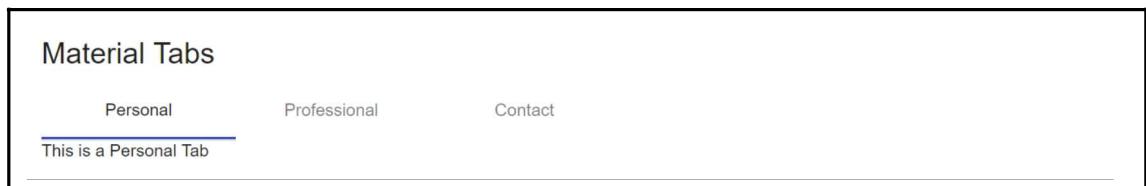


Tabs

The last layout component that we are going to learn about is tabs. Angular Material provides a module called `MatTabsModule`, which provides the `<mat-tab-group>` and `<mat-tab>` directives so that we can easily create a tabs component in our application. Take a look at the following sample code:

```
<mat-card class="z-depth" >
  <mat-card-title>Material Tabs</mat-card-title>
  <mat-card-content>
    <mat-tab-group>
      <mat-tab label="Personal"> This is a Personal Tab </mat-tab>
      <mat-tab label="Professional"> This is a Professional tab </mat-tab>
      <mat-tab label="Contact"> This is Contacts Tab </mat-tab>
    </mat-tab-group>
  </mat-card-content>
</mat-card>
```

In the preceding code, we used the `<mat-tab-group>` wrapper directive, and inside this, we use the `<mat-tab>` directive for each specific tab. Each tab will have a label that will be displayed at the top of the tab. Inside `<mat-tab>`, we will display the content of each tab. Take a look at the output of the preceding code in the following screenshot:



In the next section, we will learn about Angular Material forms. Read on.

Form controls

Forms are the main ingredient in any interactive and dynamic application. Angular Material natively supports forms and form controls that can easily be integrated into our applications. In this section, we will learn how to put together forms using Angular Material.

Forms, in general, have evolved a lot in terms of UX/UI. Angular Material supports basic form field elements that involve text fields, textareas, drop-down select options, radio buttons, and checkbox elements. Angular Material also provides advanced form elements, such as autocomplete, the datepicker, slide toggles, and so on. We will learn how to add all of this to our form as we work through our hands-on examples.

Angular Material provides a lot of modules that are related to forms and form field elements, including the following listed modules:

- MatFormFieldModule
- MatInputModule
- MatRadioModule
- MatChipModule
- MatProgressBarModule
- MatSelectModule
- MatSlideModule
- MatSlideToggleModule
- MatListModule
- MatDatePickerModule
- MatAutocompleteModule
- MatCheckboxModule

As previously mentioned, we can import these individually, or all in one go, as we did in the previous section in our `MaterialModule` file. We have our modules imported in `AppModule`; we are good to start implementing the form fields into our form. We will wrap each of the input and textarea form elements in a `<mat-form-field>` wrapper directive. To implement the input textbox, we will make use of the `matInput` attribute, along with our HTML `input` tag:

```
<mat-form-field>
<input matInput placeholder="Enter Email Address" value="">
</mat-form-field>
```

That was very simple and straightforward, right? You bet it is. Now, similarly, we can easily add a `textarea` field to our form:

```
<mat-form-field class="example-full-width">
<textarea matInput placeholder="Enter your comments here"></textarea>
</mat-form-field>
```

Okay, so it wasn't rocket science to add the `Input` and `Textarea` form elements. Next, we are going to implement a radio button and checkbox field element:

```
<mat-radio-group>
<p>Select your Gender</p>
<mat-radio-button>Male</mat-radio-button>
<mat-radio-button>Female</mat-radio-button>
</mat-radio-group>
```

To implement a radio button in our form, we will use the `<mat-radio-button>` directive. In most cases, we will also use multiple radio buttons in order to provide different options. That's where we will use a `<mat-radio-group>` wrapper directive. Similar to the radio button, Material provides a directive that we can easily use to integrate checkboxes into our application. We will make use of the `<mat-checkbox>` directive as follows:

```
<mat-checkbox>
  Agree to Terms and Conditions
</mat-checkbox>
```

The directive is provided by the `MatCheckboxModule` module, and provides a lot of properties that we can use to extend or process the data.

To implement the drop-down options in our form, we will need to use the HTML `<select>` and `<option>` tags. The Material library provides directives that we can easily use to extend the `is` capability in our form:

```
<mat-form-field>
  Select City
  <mat-select matNativeControl required>
    <mat-option value="newyork">New York City</mat-option>
    <mat-option value="london">London</mat-option>
    <mat-option value="bangalore">Bangalore</mat-option>
    <mat-option value="dallas">Dallas</mat-option>
  </mat-select>
</mat-form-field>
```

In the preceding code, for using the `<select>` and `<option>` tags, we'll be using the `<mat-select>` and `<mat-option>` directives. We are making very good progress here.

Let's keep the momentum going. The next form field element that we are going to implement is a slider component.

Sliders can be really helpful when the user wants to specify a start value and an end value. It improves the user's experience when they can just start scrolling through the range, and the data gets filtered based on the selected range. To add a slider to our form, we will need to add the `<mat-slider>` directive:

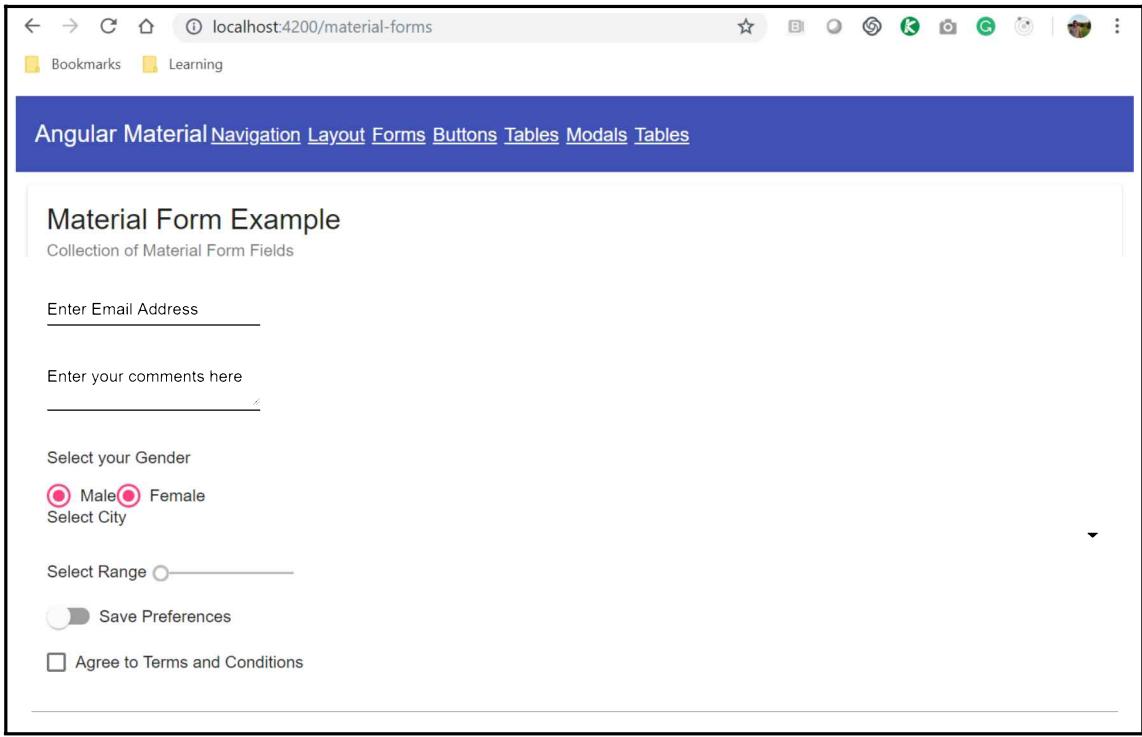
```
<mat-form-field>
  Select Range
  <mat-slider></mat-slider>
</mat-form-field>
```

That was very simple. The `MatSliderModule` API provides a lot of options to extend and use the directive in many useful ways. We can specify maximum and minimum ranges. We can set interval values, and much more. Talking about slider functionality in the UI, there is a component that we can use, called a slide toggle. We can implement a slide toggle using the `<mat-slide-toggle>` directive:

```
<mat-slide-toggle>Save Preferences</mat-slide-toggle>
```

We made use of the `<mat-slide-toggle>` directive that was provided by the `MatSlideToggleModule` module. The API provides a lot of properties, such as `dragChange`, `toggleChange`, setting color or validation as required, and so on.

Now that we have put together all of the preceding form field elements in our template file, let's run the app to see the output. Run the app using the `ng serve` command, and we should see the output shown in the following screenshot:



In the next section, we will learn about the buttons and indicator components that are provided by Angular Material.

Buttons and indicators

A quick bit of trivia here—ave you seen any website or application without any sort of buttons? If you have, please write to me.

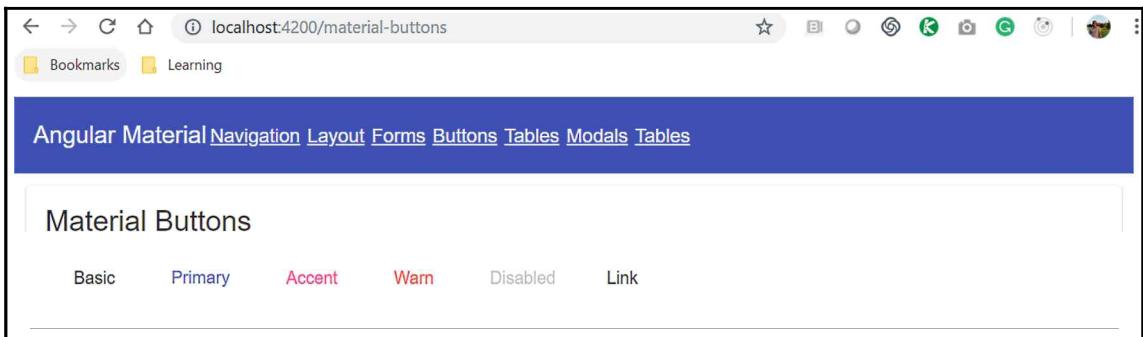
As far as my experience goes, buttons are an integral part of web applications. In this section, we will learn all about buttons, button groups, and indicators.

Angular Material provides a lot of useful and easy attributes that we can attach to the button tags, and, voila. Magic happens. The simplest way to start using an Angular Material button is by adding a `mat-button` attribute to the `<button>` tag:

```
<div>
<button mat-button>Simple Button</button>
<button mat-button color="primary">Primary Button</button>
<button mat-button color="accent">Accent Button</button>
```

```
<button mat-button color="warn">Warn Button</button>
<button mat-button disabled>Disabled</button>
<a mat-button routerLink=".">Link</a>
</div>
```

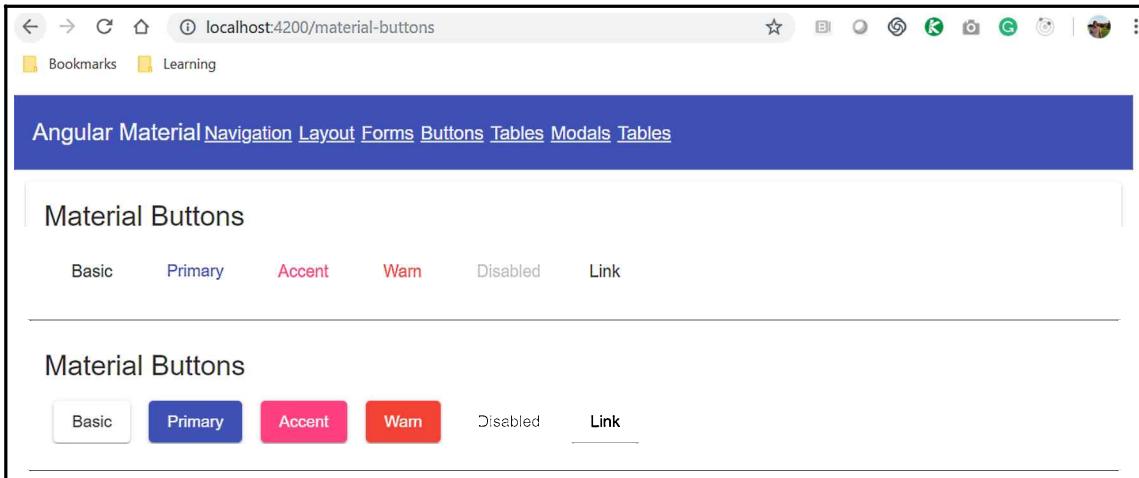
In the preceding code, we added the `mat-button` attribute to all the buttons that we have added to our `material-button.component.html` template file. We also customized the look, feel, and the behavior of the buttons using properties such as `color` and `disabled`. The output of the preceding code is displayed as follows:



The buttons in the preceding screenshot appear more like links and not buttons, right? Let's customize them to make them look more like buttons. We can easily do this by adding the `mat-raised-button` attribute. Notice that in the previous example, we used the `mat-button` attribute, and in this example, we are adding `mat-raised-button`. The updated code is as follows:

```
<div>
  <button mat-raised-button>Basic Button</button>
  <button mat-raised-button color="primary">Primary Button</button>
  <button mat-raised-button color="accent">Accent Button</button>
  <button mat-raised-button color="warn">Warn Button</button>
  <button mat-raised-button disabled>Disabled Button</button>
  <a mat-raised-button routerLink=".">Link</a>
</div>
```

The output of the preceding code is as follows. Notice the difference in the look and feel of the buttons now that the new attribute has been added:



They are pretty buttons! Using the predefined attributes allows us to maintain the uniformity of the buttons across the application.

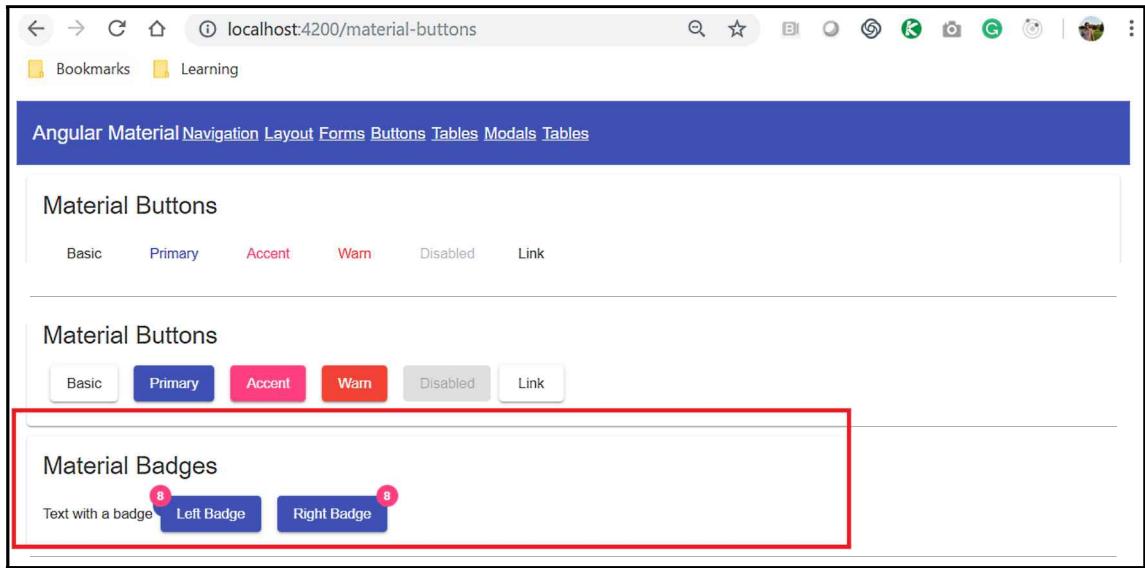
Next, we are going to explore the indicators provided by Angular Material. We will learn about badges and progress bar components as part of the indicator components.

Badges are a way to highlight some data along with other UI elements. We may come across use cases where we want to use badges along with buttons. You must already be thinking, can we also add some UX for the buttons to design some functionality, too? Yes, we can!

Angular Material provides a module called `MatBadgeModule`, which has implementations for the `matBadge`, `matBadgePosition`, and `matBadgeColor` attributes, which can easily be used to set badges to the buttons. Take a look at the following sample code:

```
<button mat-raised-button color="primary"  
       matBadge="10" matBadgePosition="before" matBadgeColor="accent">  
  Left Badge  
</button>
```

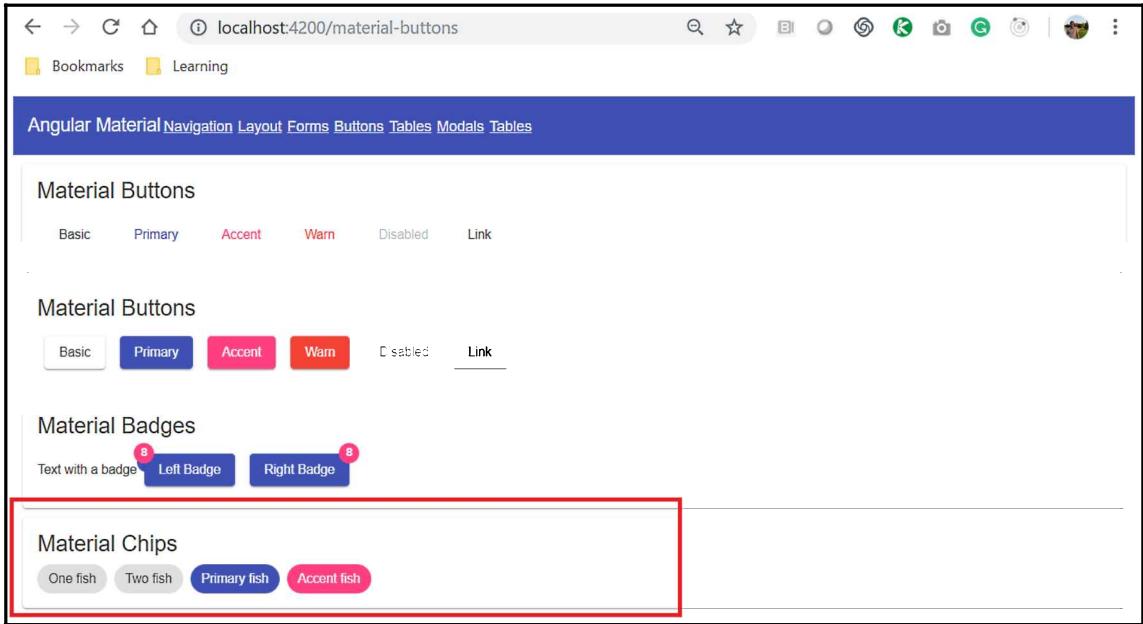
In the preceding code, we added a button element, and we specified the attributes, such as `matBadge`, `matBadgePosition`, and `matBadgeColor`. The output of the preceding code is as follows:



This was a button with badges. There is another UI component called chips. We can easily use these in order to enhance the UX as well. Think of material chips as *tags* in any other application you have used previously. Angular Material provides a module called `MatChipModule`, which provides the `<mat-chip-list>` and `<mat-chip>` directives, which we can easily integrate into our application. Take a look at the following sample code:

```
<mat-chip-list>
<mat-chip color="primary" selected>New York</mat-chip>
<mat-chip>London</mat-chip>
<mat-chip>Dallas</mat-chip>
<mat-chip>Accent fish</mat-chip>
</mat-chip-list>
```

In the preceding code, we used the directives resulting from `MatChipModule`, and from putting together the tags. The output of the preceding code is as follows:

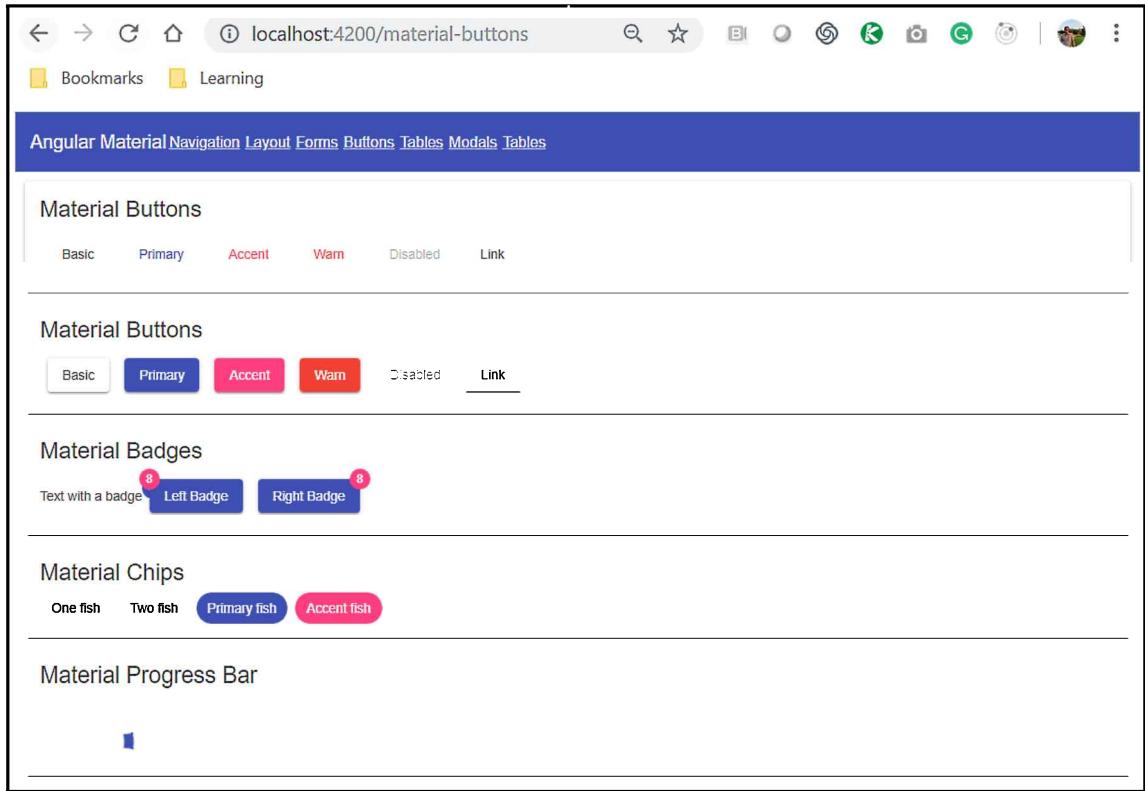


That was neat. The next indicator that we will learn to implement is a very important one; the progress bar. We need to show and inform our users about actions that are being performed in the background, or show the progress of processing some user data. In such situations, we will need to clearly show this using a progress bar.

Angular Material provides modules named `MatProgressBarModule` and `MatProgressSpinnerModule`, using which, we can easily add a loading icon or spinner to our web application. Using the API properties and events, we can easily capture and process the data as well. Take a look at the following sample code:

```
<mat-spinner></mat-spinner>
```

That's it? Really? Are we kidding? No, we are not. Just by using this module, we should see the spinning wheel displayed in our application. Take a look at the output of the preceding code:



In the next section, we will learn all about the modals and dialog windows that are provided by Angular Material.

Popups and modals

Modern web applications have introduced a lot of innovative UX features and functionalities. One feature that really stands out has to be modal windows. Take any major web application; it will have some flavor of modal window implemented in it. The Angular Material library, too, provides us with an easy way to implement modal or dialog pop-up windows.

Angular Material has a module named `MatDialogModule`, which provides various classes that we can use in our component classes. Unlike other UI components, there are no directives that you can directly use in the template file; instead, we need to achieve this functionality programmatically. Before we get into creating our dialog window implementation, we will need a component in which we can store the modal window content. Run the following command and generate a component. Let's call it the `addDialog` component:

```
ng g c addDialog
```

When the command is executed successfully, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter9_material\AngularMaterial>ng g c addDialog
CREATE src/app/add-dialog/add-dialog.component.html (29 bytes)
CREATE src/app/add-dialog/add-dialog.component.spec.ts (650 bytes)
CREATE src/app/add-dialog/add-dialog.component.ts (285 bytes)
CREATE src/app/add-dialog/add-dialog.component.scss (0 bytes)
UPDATE src/app/app.module.ts (2866 bytes)
```

Now, open the newly created `add-dialog.component.html` file, and add some content. Even *Hello World* is just fine for now.

Next, let's start modifying our `MaterialModalComponent` class, and add the following code into it:

```
import { Component, OnInit, Inject } from '@angular/core';
import { VERSION, MatDialogRef, MatDialog } from '@angular/material';
import { AddDialogComponent } from '../add-dialog/add-dialog.component';

@Component({
  selector: 'app-material-modals',
  templateUrl: './material-modals.component.html',
  styleUrls: ['./material-modals.component.scss']
})
export class MaterialModalsComponent implements OnInit {

  constructor(private dialog: MatDialog) { }

  ngOnInit() { }

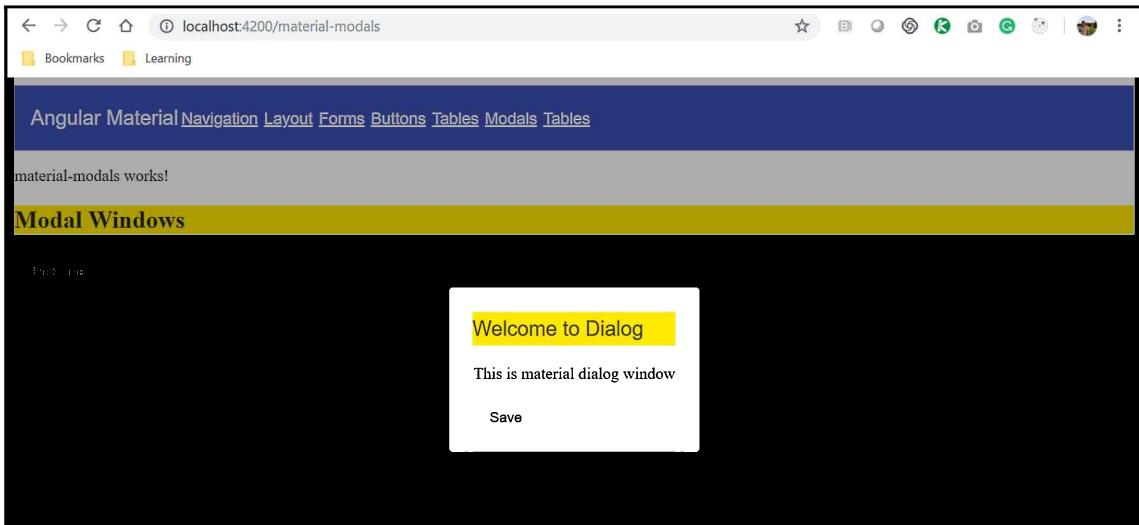
  openDialog() {
    const dialogRef = this.dialog.open(AddDialogComponent);
  }
}
```

Let's analyze the preceding code. We imported all the required modules into the file. We then imported VERSION, MatDialogRef, and MatDialog into our component class. We also imported AddNewComponent, which we want to display in the modal window. Since we imported MatDialog into the class, we need to inject it into our constructor method, and then create an instance of it. We will then create another method, named openDialog. In this method, by using the MatDialog instance, we are calling the method open and passing AddNewComponent as the parameter. We have implemented the functionality of the modal window, but this won't work until we actually call the openDialog method.

So, let's open our material-modal.component.html template file, and add the following line to it:

```
<button mat-raised-button (click)="openDialog()">Pick one</button>
```

There's not much to describe here. We just added a button and attached an onclick event in order to call the openDialog method: simple and sweet. Let's run the app using the ng serve command, and we should see the following output:



In my `AddDialogComponent`, I have added some text and a button. You can add or design your own template as well. The API provides a lot of properties and events that we can associate with the dialog window.

In the next section, we will learn about the data tables feature that is provided by Angular Material.

Data tables

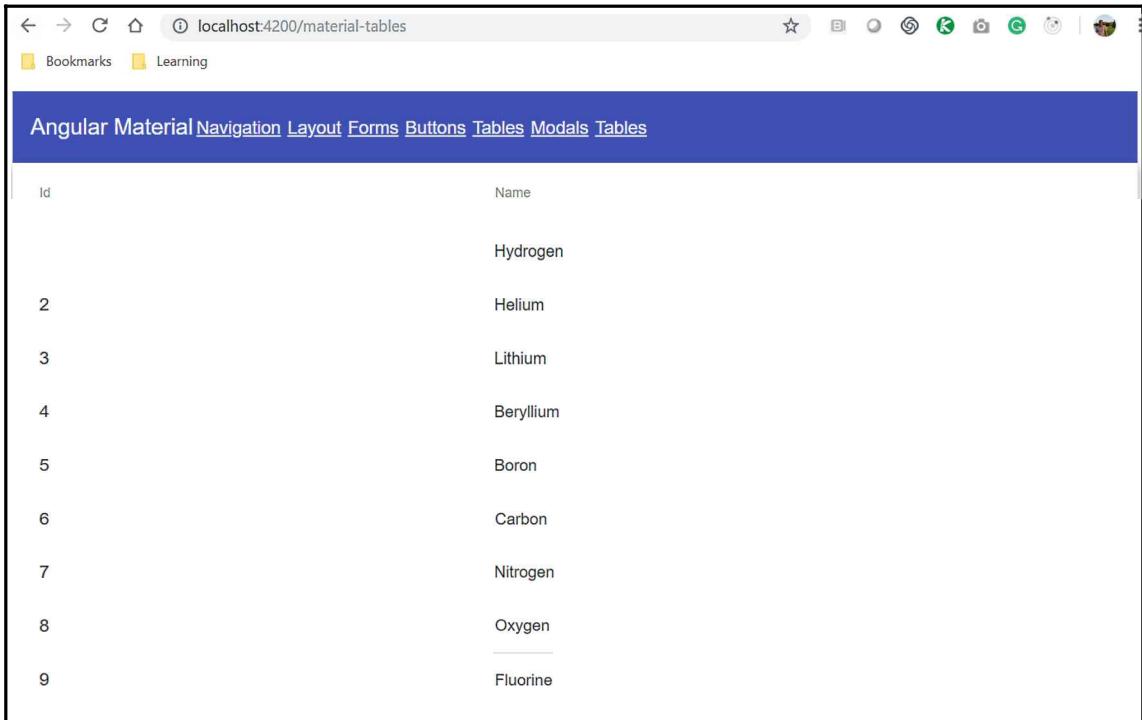
Tables are one of the key aspects of designing complex behind-the-login screen functionalities. I say behind the login screen, because that way, the search engine optimization debate won't come into the picture. The problem with traditional tables is that we need to map the data, rows, and columns ourselves, along with implementing pagination and responsiveness. Thanks to Angular Material, we can now have all of that generated for us with just one line of command. That's right, you read it correctly—with only one command, when we use schematics. Run the following command, and we should have our data table ready in no time:

```
ng generate @angular/material:table issueList
```

We use the `ng` command to specify to the schematics with which we want to generate the table from Angular Material, and that it should be created in a new component named `issueList`. Upon the successful running of the command, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter9_material\AngularMaterial>ng generate @angular/material:table issueList
CREATE src/app/issue-list/issue-list.datasource.ts (3379 bytes)
CREATE src/app/issue-list/issue-list.component.html (884 bytes)
CREATE src/app/issue-list/issue-list.component.spec.ts (941 bytes)
CREATE src/app/issue-list/issue-list.component.ts (721 bytes)
CREATE src/app/issue-list/issue-list.component.css (37 bytes)
UPDATE src/app/app.module.ts (3042 bytes)
```

Run the app using the `ng serve` command, and navigate to the route of the table. We should see the output shown in the following screenshot:



A screenshot of a web browser window displaying a dynamic table. The address bar shows `localhost:4200/material-tables`. The browser interface includes standard navigation buttons, a search field, and a toolbar with various icons. The main content area has a blue header bar with the text "Angular Material" followed by links for Navigation, Layout, Forms, Buttons, Modals, and Tables. Below this is a table with two columns: "Id" and "Name". The data rows are as follows:

| Id | Name |
|----|-----------|
| 2 | Hydrogen |
| 3 | Helium |
| 4 | Lithium |
| 5 | Beryllium |
| 6 | Boron |
| 7 | Carbon |
| 8 | Nitrogen |
| 9 | Oxygen |
| | Fluorine |

Voila! We now have our dynamic table ready to use. We can customize the data source values and the columns that we need to display and update the pagination just using the configurations in our component class. Go ahead and give it a try.

Summary

We started this chapter by creating the placeholder components for each of the main categories of the UI components. The components are categorized under various category layouts, material cards, form controls, navigations, buttons and indicators, modals and popups, and tables.

We started by creating the navigation menu component. We learned how to autogenerate the navigation menu component using schematics. We then also learned how to implement a custom menu for our apps. Next, we started learning and implementing the layout components that are provided by Angular Material. In the layout components, we learned about Material cards. We learned how to include various content inside the Material cards. We learned about various lists that are supported by Material. We learned about lists with dividers, and navigation lists. We also learned how to implement accordions and expansion panels to better group and arrange the data. We also explored how to use the stepper component, which is very useful when designing UX for data that requires various steps. Along the same lines, we learned about grouping things using tabs.

Next, we explored Material forms and learned how to implement form field elements including input, textarea, radio and checkbox buttons, sliders, and slide toggles. We also learned about different types of buttons and indicators, including badges and tags that are provided by Material. We then learned about and implemented the modals and pop-up windows that are provided by Angular Material.

Finally, we learned about data tables, and how schematics help us to set up data tables quickly in our applications.

A separate book is required if we want to cover every nook and hook of the Angular Material components. We have tried to give you an overview of the different components available, and why you might consider Material in your next project when it makes sense to do so and suits you/your clients. It's definitely worth a try!

10

Working with Forms

Let's start this chapter with a simple guessing game. Can you think of any web application that does not have any sort of form, such as, signing up, login, create, contact us, edit forms, and so on; the list is endless. (wrong answer—even the Google homepage has a search form.)

Technically, it's possible. I am 100% sure there are some websites that may not use forms at all, but I am also equally confident they will be static and won't interact or engage with users dynamically, which brings us to the main context and focus of this chapter: implementing and using forms in our Angular application.

OK, let's now take a look at what we'll be covering in this chapter:

- Introduction to Bootstrap forms
- Bootstrap form classes
- Bootstrap form classes—extended
- Angular forms
- Template-driven forms
- Reactive forms
- Form validations
- Submitting and processing form data

Bootstrap forms

We will learn to make use of the awesome Bootstrap library, which offers a rich set of classes and utilities for us to design and develop forms in our applications, making developers' and designers' life easy!

What are forms?

Forms are sets, collections of input fields gathered together to enable us to collect data from the user through the keyboard, mouse, or touch input.

We will learn to stitch input elements together and build some sample forms, such as logging in, signing up, or for when the user forgets their password.

Before we jump into creating the forms, here's a quick list of available HTML input elements that we can use in our apps:

- Input (including text, radio, checkbox, or file)
- Textarea
- Select
- Button
- Form
- Fieldset



If you want a quick refresher tutorial on HTML tags and elements, you can visit [w3schools.com](https://www.w3schools.com/html/html_form_elements.asp).

Armed with the knowledge about forms and the available HTML elements, it's hands-on time.

Bootstrap form classes

In this section, we will learn about the available classes from the Bootstrap framework, which we can make use of while building our forms. Each form can consist of various input elements, such as textual form controls, file input controls, input checkboxes, and radio buttons. The `.form-group` class is an easy way to add structure to our forms. Using the `.form-group` class, we can easily group input elements, labels, and help text to ensure proper grouping of elements in the form. Inside the `.form-group` element, we will add input elements and assign them each the `.form-control` class.

A sample of a grouping of elements using the `.form-group` class is as follows:

```
<div class="form-group">
<label for="userName">Enter username</label>
<input type="text" class="form-control" id="userName" placeholder="Enter
username">
</div>
```

In the preceding code, we are creating a form group consisting of label and input elements of type `text`.

On the same lines, we can easily add the textual input elements, such as `email`, `password`, and `textarea`. The following is the code to add the input element of type `email`:

```
<div class="form-group">
<label for="userEmailAddress">Enter email address</label>
<input type="email" class="form-control" id="emailAddress"
placeholder="name@example.com">
</div>
```

Similarly, we can easily add an input element of type `password` as well. Again, notice that we are making use of `form-group` as a wrapper and adding `form-control` to the element:

```
<div class="form-group">
<label for="userPassword">Enter password</label>
<input type="password" class="form-control" id="userPassword">
</div>
```

Nice. We learned to use `form-group` and `form-control` classes on input elements. Now, let's add the same classes to the `textarea` element. The following is the sample code for adding the classes to a `textarea` element:

```
<div class="form-group">
<label for="userComments">Example comments</label>
<textarea class="form-control" id="userComments" rows="3"></textarea>
</div>
```

You will notice that all of the preceding elements have the same structure and grouping. For `select` and `multiple select` input elements, it's also exactly the same.

In the following sample code, we are creating a `select` drop-down element and using the `form-control` class:

```
<div class="form-group">
<label for="userRegion">Example select</label>
<select class="form-control" id="userRegion">
<option>USA</option>
```

```
<option>UK</option>
<option>APAC</option>
<option>Europe</option>
</select>
</div>
```

We have added a `select` drop-down element and will allow the user to select only one option from the list. And just by adding an additional attribute, `multiple`, we can easily allow the user to select multiple options:

```
<div class="form-group">
<label for="userInterests">Example multiple select</label>
<select multiple class="form-control" id="userInterests">
<option>Biking</option>
<option>Skiing</option>
<option>Movies</option>
<option>Music</option>
<option>Sports</option>
</select>
</div>
```

That was simple and straightforward. Let's keep rolling.

Now, let's proceed to other important input elements: checkboxes and radio buttons. However, the classes are different for `checkbox` and `radio` elements.

There are three new classes that we will learn to implement for `checkbox` and `radio` elements:

- To wrap the element, we will use the `form-check` class
- For the `input type checkbox` and the `radio` element, we will use `form-check-input`
- For `checkbox` and `radio` elements, we will need to display labels, for which we will use the `form-check-label` class:

```
<div class="form-check">
<input class="form-check-input" type="checkbox" value="" id="Worldwide">
<label class="form-check-label" for="Worldwide">
Worldwide
</label>
</div>
```

In the preceding code, we are using the `.form-check` class, `.form-check-input`, and `.form-check-label` to our wrapper `div` and `label` elements.

Ditto, on similar lines, we will use the preceding classes to add to `input radio` elements:

```
<div class="form-check">
  <input class="form-check-input" type="radio" name="gender" id="maleGender"
    value="option1" checked>
  <label class="form-check-label" for="maleGender">
    Male
  </label>
</div>
<div class="form-check">
  <input class="form-check-input" type="radio" name="gender"
    id="femaleGender"
    value="option2">
  <label class="form-check-label" for="femaleGender">
    Female
  </label>
</div>
```

In the preceding code, we are creating two radio buttons for the user to select their gender, and the user can make only one selection out of the two options.

In most modern web applications, we will need a user to be able to upload files or assets to our applications. Bootstrap provides us class named "form-control-file", which we can associate to the file upload element.

We will use the `form-control-file` class to our `input type file` element. The sample code for this is as follows:

```
<div class="form-group">
  <label for="userProfilePic">Upload Profile Pic</label>
  <input type="file" class="form-control-file" id="userProfilePic">
</div>
```

Great. We have learned to put together all the elements with which we can create our beautiful and powerful forms.

Bootstrap form classes – extended

We have learned to create forms with input elements and add some of the available form classes in Bootstrap to group elements, as well as to improve our application.

In this section, we will look at other additional classes and attributes provided by the Bootstrap framework, which can be used to improve **user experience (UX)**, as well as extend the behavior of the elements:

- Sizing
- Readonly
- Inline forms
- Forms using Bootstrap grid classes
- Disabled
- Help text
- Plain text inside `form-group`

We will go through each one of the aforementioned options and learn to implement them and see them in action.

Sizing

We can set the size of the input elements in our form. We can control the height of the element using various classes for small, medium, and large resolutions.

We have learned to use the `.form-control` class in the previous section and, by default, the medium size height with the `.form-control-md` class is applied. There are other classes available to set the height as large or small. We can use `.form-control-lg` and `.form-control-sm`, respectively.

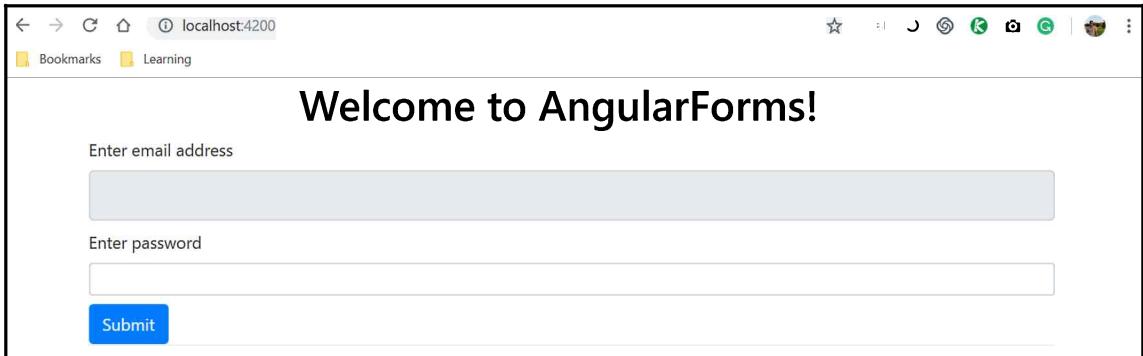
The following is the sample code, where we are setting the height of the email address element as large using the `.form-control-lg` class, and the password field with the `.form-control-sm` class:

```
<form>
  <div class="form-group mb-2 mr-sm-2">
    <label for="userEmailAddress">Enter email address</label>
    <input type="email" class="form-control form-control-lg" id="userEmailAddress">
  </div>
  <div class="form-group mb-2 mr-sm-2">
    <label for="userPassword">Enter password</label>
    <input type="password" class="form-control form-control-sm" id="userPassword">
  </div>

  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

We have added the `form-control-lg` and `form-control-sm` classes to the form control's email address and password form elements, respectively.

When we run the application, the output of the preceding code is as follows:



In the preceding screenshot, notice the difference in heights of the input elements. The email address text field has increased in height and the password field is small.

Readonly

We may come across a use case where we will need to disable a field and make it read-only. We can utilize the attribute `readonly`. By adding the Boolean `readonly` attribute to any form control element, we can disable that element.

The sample code showing the usage of the `readonly` attribute on the `username` field is as follows:

```
<div class="form-group">
  <label for="userName">Enter username</label>
  <input type="text" class="form-control" id="userName" placeholder="E.g
    packtpub" readonly
```

The output of the preceding code is displayed as follows. Notice that **email address** field is disabled, and so the user will not be able to add/edit the element:



Inline forms

The design is also an equally important aspect of how we display the form. We may come across a use case where we have a requirement to place our form horizontally, instead of the regular vertical way.

Bootstrap has the `.form-inline` class to support inline or horizontal forms. When the `.form-inline` class is used, the form elements automatically float horizontally.

The following is some sample code, where we create the login form with an email address and password. We make it inline using the `form-inline` class :

```
<form class="form-inline">
  <div class="form-group">
    <label for="userEmailAddress">Enter email address</label>
    <input type="email" class="form-control" id="emailAddress"
      placeholder="name@example.com">
  </div>

  <div class="form-group">
    <label for="userPassword">Enter password</label>
    <input type="password" class="form-control" id="userPassword">
  </div>
</form>
```

In the preceding code, the important thing to note is the usage of the `.form-inline` class.

The output of the preceding code is displayed as follows:



By default, all forms designed using Bootstrap are vertical.



Forms using Bootstrap grid classes

Remember the Bootstrap grid classes we learned about in chapter 3, *Bootstrap – Grid Layout and Components*? Yes, rows, columns, and designing the layout of the screen.

In this section, we will learn to use the same row and column grid classes inside our forms, which is good news because using these classes, we can design a custom layout and update the look and feel of the form.

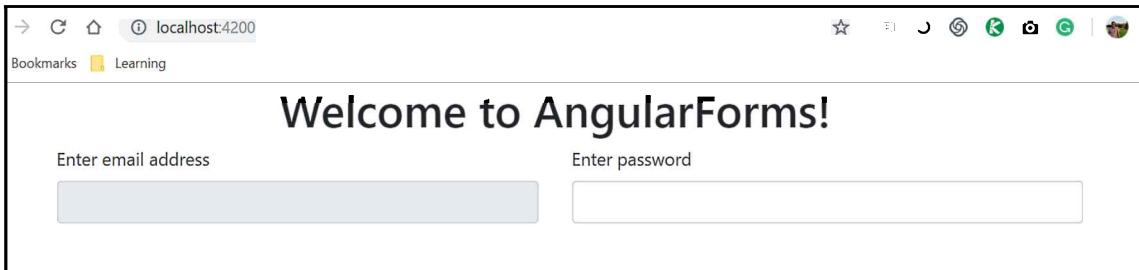
The sample code for this is as follows:

```
<form>
  <div class="row">
    <div class="col">
      <label for="userEmailAddress">Enter email address</label>
      <input type="email" class="form-control" id="emailAddress" readonly>
    </div>
    <div class="col">
      <label for="userPassword">Enter password</label>
      <input type="password" class="form-control" id="userPassword">
    </div>
  </div>
</form>
```

In the preceding code, instead of using the `.form-group` class, we are making use of the `row` and `col` classes, which are mainly used for designing layout.

We make a single row with two columns, and in each column, we add input elements.

The output of the preceding code is as follows:



Here's your homework now. Try out these fun use cases using grid classes with forms:

- Add more input elements in the same row by adding more column div elements to the same row
- Add multiple rows to the form
- Assign a fixed width for some columns (column 4 or column 3)

Disabled

While developing web applications with critical and complex compliance requirements, it's very common that we will have to disable certain input elements based on user selections.

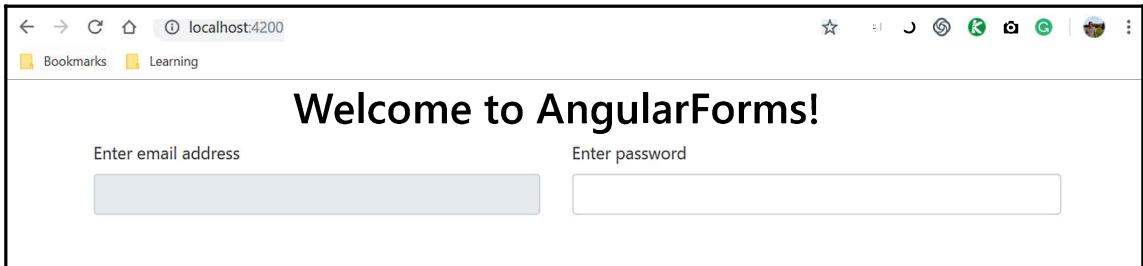
A good use case is where some fields are not applicable to a specific country the user has selected, so we need to disable other dependent fields.

Using the `disabled` attribute which takes a Boolean value, we can either disable a form or a particular element.

Let's see the `disabled` attribute in action:

```
<form>
  <div class="row">
    <div class="col">
      <label for="userEmailAddress">Enter email address</label>
      <input type="email" class="form-control" id="emailAddress" disabled>
    </div>
    <div class="col">
      <label for="userPassword">Enter password</label>
      <input type="password" class="form-control" id="userPassword">
    </div>
  </div>
</form>
```

In the preceding code, we are using the `disabled` attribute. We can see in the following screenshot that the email address field is completely disabled:



We can make any element disabled just by adding the `disabled` attribute to the element. This is good, but what if we want to disable the entire form in one go? We can do that as well.

Take a look at the following code:

```
<form>
  <fieldset disabled>
    <div class="row">
      <div class="col">
        <label for="userEmailAddress">Enter email address</label>
        <input type="email" class="form-control" id="emailAddress">
      </div>
      <div class="col">
        <label for="userPassword">Enter password</label>
        <input type="password" class="form-control" id="userPassword">
      </div>
    </div>
  </fieldset>
</form>
```

We are adding the `fieldset` tag inside the `form` to wrap all the elements of the form together and apply the `disabled` attribute to the `fieldset` element, which will disable the entire form in one go.

The output of the preceding code is displayed as follows:



Help text inside forms

Any good web application will have beautiful, yet powerful forms, which talk to users and create a good UX experience.

Help text is one of the options we have to notify the user about any errors, warnings, or mandatory fields in the form, so that the user can take necessary actions.

Take a look at the following code:

```
<form>
  <div class="form-group">
    <label for="userEmailAddress">Enter email address</label>
    <input type="email" class="form-control" id="userEmailAddress">
    <small id="userEmailAddressHelp" class="form-text text-danger">
      Email address cannot be blank.
      Email address should be atleast 3 characters
    </small>
  </div>
  <div class="form-group">
    <label for="userPassword">Enter password</label>
    <input type="password" class="form-control" id="userPassword">
  </div>
</form>
```

In the preceding code, we are adding text inside the `<small>` tag, and assigning the `.form-text` class and `.text-danger`.

The output of the preceding code is as follows:



Displaying input elements as plain text

We may come across a requirement where we need to display an input element as just text, and not as an input element.

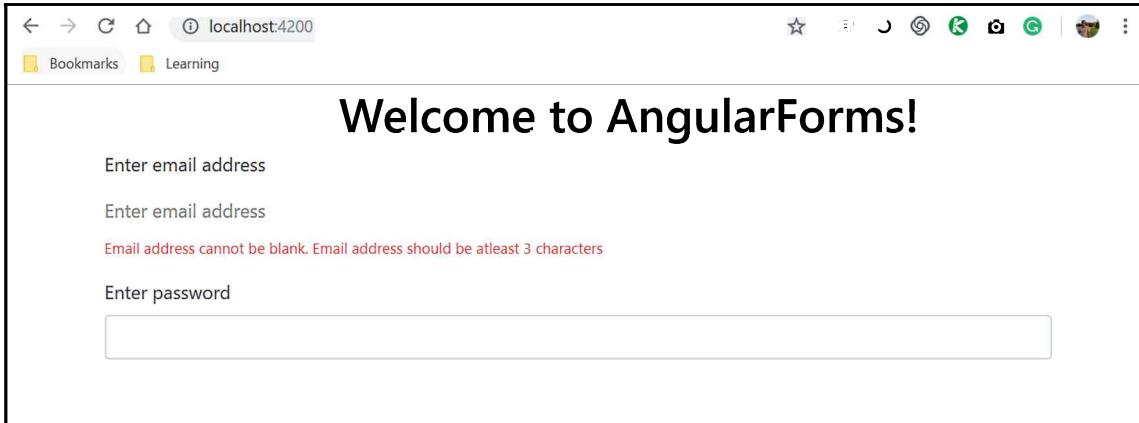
We can do this simply by customizing our style sheet, or just by using the `.form-control-plaintext` class inside the element with the `.form-group` class.

Take a look at the following code:

```
<form>
  <div class="form-group">
    <label for="userEmailAddress">Enter email address</label>
    <input type="email" class="form-control-plaintext" id="userEmailAddress"
      placeholder="Enter email address">
    <small id="userEmailAddressHelp" class="form-text text-danger">
      Email address cannot be blank.
      Email address should be atleast 3 characters
    </small>
  </div>
  <div class="form-group">
    <label for="userPassword">Enter password</label>
    <input type="password" class="form-control" id="userPassword">
  </div>
</form>
```

In the preceding code, we have added the `.form-control-plaintext` class to the input element.

The output of the preceding code is as follows:



In this section, we have learned about various classes and attributes that we can use to enhance and make our forms more interactive and powerful, and above all, add to better user design and experience.

Angular forms

In this section comes the real action of forms in Angular applications. Forms are at the heart of any application and are the main building blocks to gather, view, capture, and process data provided by the users. We will continue to use the Bootstrap library to enhance the design of our forms in this section.

Angular provides two different approaches to building forms inside our apps.

The two approaches provided by Angular for building forms are as follows:

- Template-driven forms: HTML and data binding defined in template files
- Reactive or model-driven forms, using model and validation in the Component class files



Although the form model is a commonality between template-driven forms and reactive forms, they are created differently.

The main difference between reactive forms and template-driven forms, when it comes to the template, is in the data binding. In template-driven forms, we use two-way data binding to bind our data model directly to the form elements. On the other hand, with reactive forms, we bind our data model to the form itself (as opposed to its individual form elements).

We will explore each of these approaches in detail, learn about the pros and cons of the approaches, and finally, we will build some forms using both approaches. Let's roll.

Template-driven forms

Template-driven forms, as the name suggests, involve all the heavy-duty work of forms being performed right inside the template of the component. The approach is good and is recommended when dealing with simple, straightforward forms, without much complex validation or rules.

All the logic is in the template files, which essentially means that we will make use of HTML elements and attributes. With template-driven forms, we use HTML to create the form and the input elements and create validation rules as HTML attributes. Two-way data binding is the key part, so we can tie the form elements to the properties in the Component class.

Angular automatically tracks the form and input element states by generating the form model automatically for us to use. We can directly take the form as an object and process data easily.

When using the template-driven approach, we first import the `FormsModule`, so we can have access to the following directives:

- `ngForm`
- `ngModel`
- `ngModelGroup`



We need to import the `FormsModule` into our `app.module.ts` file.

Let's take a look at the pros and cons of using the template-driven form approach in our apps.

Template-driven forms – pros

Template-driven forms can be very useful and helpful if the forms in our application are simple and straight forward with less metadata and validations. In this section we will highlight the pros of using template-driven forms in our applications:

- Template-driven forms are very easy to use
- Suitable for simple and straightforward use cases
- Easy-to-use two-way data binding, so there is minimal code and complexity
- Angular automatically tracks the form and input element state (you can disable the **Submit** button if the form's state is not complete)
- Not recommended if the form has complex form validations or require custom form validations

Template-driven forms – cons

In the previous section, we have learned about the advantages of using template-driven forms in our application and we have made a strong argument about the pros of using the template-driven form approach. In this section, we will learn about some of the cons of using template-driven forms in our applications:

- Not recommended or suitable where the requirements of the form are complex and comprise custom form validations
- Unit testing cannot be fully covered to test all use cases

Template-driven forms – important modules

Armed with knowledge about the pros and cons of using the template-driven approach, without wasting any time we will deep dive into learning how to implement the template-driven forms in our application. We will start by learning about the required modules and gradually progress to create forms in our application. As explained in the preceding sections, template-driven forms are mostly defined in the template file. Before we jump into creating examples of template-driven forms, we should understand some of the most important concepts related to forms, namely, `ngForm` and `ngModel`:

- `ngForm`: This is the directive that helps to create the control groups inside the form directive
- `ngModel`: When `ngModel` is used on elements inside `ngForm`, all the elements and data get registered inside `ngForm`



If the Angular form is using `ngForm` and `ngModel`, it means that the form is template-driven.

Building our login form

So far, we have a good high-level understanding of what template-driven forms are. In this section, we will put our knowledge to work by building a form. Let's put together a form using the classes we have learned in the preceding section.

The use case we will work on is the user login form for our application. First, we need to generate our login component. Run the following `ng` command to generate the login component:

```
ng g c login
```

The output of the preceding command is displayed as follows:



We will need to add our route path in the `app-routing.module.ts` file in order to access the routes for `login` and `register`.

We are building our form using the template-driven approach, so we will need to do most of the work in our template file. Before we start modifying our template file, we will need to import a required module into our `app.module.ts` file.

Open the `app.module.ts` file and add the following line of code:

```
import {FormsModule} from '@angular/forms';
```

Once we have imported `FormsModule` into our `app.module.ts` file, don't forget to add it to our list of imports inside `ngModule`.

The updated `app.module.ts` file is displayed as follows:

```
import { RegisterComponent } from './register/register.component';

import {FormsModule} from '@angular/forms';
import { FormGroup, FormControl,Validators, NgForm } from '@angular/forms';

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

Now, let's open our login component template file and create our login form in the `login.component.html` file. The following is the code we will add to our template file:

```
<form #loginForm="ngForm" (ngSubmit)="login(loginForm.value)">
  <h3 class="text-center text-primary">Login</h3>
  <div class="form-group">
    <label for="username">Username:</label><br>
    <input type="text" [ngModel]="username" name="username"
      class="form-control">
  </div>
  <div class="form-group">
    <label for="password">Password:</label><br>
    <input type="password" [ngModel]="password" name="password"
      class="form-control">
  </div>

  <button type="submit" class="btn btn-primary">Sign in</button>
</form>
```

Let's analyze the preceding code in depth. We are creating a form using the HTML input elements and adding a username, password, and Submit button to the form. Important things to note are that for the form itself, we are telling the template that the form is `ngForm` and `ngForm` will group all the input elements of the form together into the `#loginForm` template variable. For the input elements, we have added the `ngModel` attribute and we specify the `name` attribute for the elements.

Using `ngForm`, we can now easily retrieve the value of the elements inside the form. Since we have defined the local `#loginForm` template variable we can now use its properties easily. `loginForm` has the following properties:

- `loginForm.value`: Returns the object containing all the values of the input elements inside the form
- `loginForm.valid`: Returns if the form is valid or not, based on the HTML attribute validators applied in the template
- `loginForm.touched`: Returns `true` or `false` depending on whether the form was touched/edited by the user or not

In the preceding code, we are passing `loginForm.value` to the component. We can pass any of these value to the component for processing or validation. Notice that we are also calling a `login` method, which we need to implement in our Component class file.

Now, let's create a method in our Component class to capture the data coming in from our `loginForm`. We are collecting the value of the form and displaying it in the console:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {

  constructor() { }

  login(loginForm) {
    console.log(loginForm);
    console.log(loginForm.controls.username);
  }
}
```

Run the app using the `ng serve` command, and we should see the output shown in the following screenshot:



Remember that in typical server-side scripting, we used to write `action` and `method` attributes for forms. We do not need to define these anymore, since they are declared and used in the `Component` class.

This is good stuff and good progress. We will continue to use the preceding login form and add validations shortly. Let's keep digging for more information.

Model-driven forms, or reactive forms

Reactive forms are also called model-driven forms. In model-driven forms, the model is created in the `Component` class file and is responsible for doing the form validation, processing data, and much more.

Angular internally builds a tree structure of the Angular form controls, which makes it much easier to push and manage data between the data models and the UI elements.

We need to build the form model in our `Component` class by creating the instances of the building blocks (that is, `FormControl` and `FormGroup`). Additionally, we write the validation rules and the validation error messages in the class as well. We even manage the properties (that is, the data model) in the class as opposed to using data binding in the HTML.

Template-driven forms put the responsibility for the forms on the template, whereas reactive forms shift the responsibility for validation to the Component class.



In this chapter, we will use the both terms: model-driven forms and reactive forms, as both refer to the same thing.

Model-driven forms – pros

Reactive forms are very useful in creating, validating and applying custom form validations to our forms in our applications. We can easily trust the model driven approach to do all the heavy duty work that is usually associated with any complex forms. In this section, we will list and understand the pros of using model-driven forms in our applications:

- Greater flexibility for more complicated validation scenarios and custom, complex form validations
- The data model is immutable
- Since the data model is immutable, no data binding is done
- It's easier to add input elements dynamically (such as subtasks on a task form) using form arrays
- It's easy to bind various events to input elements using `HostListener` and `HostBindings`
- All the code for the form controls and validations is inside the component, which makes templates much simpler and easier to maintain
- It's easier to unit test

Model-driven forms – cons

All good things in life has some form of cons attached to them. Reactive forms are no different in nature. While the pros and advantages of using reactive forms can certainly outweigh the cons but still it's important to learn and understand the cons of using reactive forms in our applications. In this section, we will list the cons of using model-driven forms in our applications:

- Immediate beginners may find the initial learning curve too high
- The developer is expected to have knowledge about the various modules required to work with model-driven forms, such as `ngvalidators`, and so on

Model-driven forms – important modules

We create the model using the two powerful classes provided by Angular—`FormGroup` and `FormControl`:

- `FormControl`: Tracks the value and state of individual form input elements
- `FormGroup`: Tracks the value and state of a group of form controls
- `FormBuilder`: Helps us to develop forms with their initial value and their validations

Just as we imported `FormsModule` in to our template-driven forms, we will need to import `ReactiveFormsModule` in to our `app.module.ts` file.

The updated `app.module.ts` file should look like the following screenshot:

```
import { RegisterComponent } from './register/register.component';

import { ReactiveFormsModule } from '@angular/forms';
import { FormGroup, FormControl, Validators, NgForm } from '@angular/forms';

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

Armed with all the knowledge about the model-driven form approach, it's time for a hands-on example.

Reactive forms – registration form example

In the previous section while covering template-driven forms, we have created our login form for our application. It's time to do a hands-on exercise using the reactive forms. The fundamental idea behind implementing login and registration forms using different approaches is to show you the difference in implementation of each approach. There is no right or wrong approach, the decision is driven by the complexity and requirement of our forms in the applications.

In this section, we will learn to implement our new user registration form using the model-driven approach.

First, we will need to generate our register component. Run the following ng command to generate the register component:

```
ng g c register
```

The output of the preceding command is as follows:

```
D:\book_2\book\chapter10_forms\AngularForms>ng g c register
CREATE  src/app/register/register.component.html (27 bytes)
CREATE  src/app/register/register.component.spec.ts (642 bytes)
CREATE  src/app/register/register.component.ts (278 bytes)
CREATE  src/app/register/register.component.scss (0 bytes)
UPDATE  src/app/app.module.ts (739 bytes)
```

Since we are talking about model-driven forms, all the hard work had to be done in the Component class. We will still need to have a template for our reactive forms, but we won't be adding any validations or data binding into the template.

We want our registration form to have four form elements—that is, fields for full names, email addresses, passwords, and terms and conditions.

Let's update our Component class in the `register.component.ts` file and create an instance of `FormGroup`:

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

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.scss']
})
export class RegisterComponent implements OnInit {

  registerForm = new FormGroup({
    fullName: new FormControl(),
    emailAddress: new FormControl(''),
    password: new FormControl(''),
    termsConditions: new FormControl('')
  });
  constructor() { }

  ngOnInit() {
  }

  register()
  {
    console.log(this.registerForm.value);
  }
}
```

You will notice a lot of new stuff in the preceding code. Let's take it slowly, step by step. We are importing the required modules, `FormGroup` and `FormControl`, from the `angular/core`. Inside the Component class, we are creating an instance of the `FormGroup` class, `registerForm`. You will notice that we are now creating multiple `FormControl` instances, each one for a form element that we want to add to our form.

Is that all we need to do? For now, yes. Remember, as explained before, that reactive forms also need to have a basic template, but all the logic and validations will be inside the component, rather than the template file.

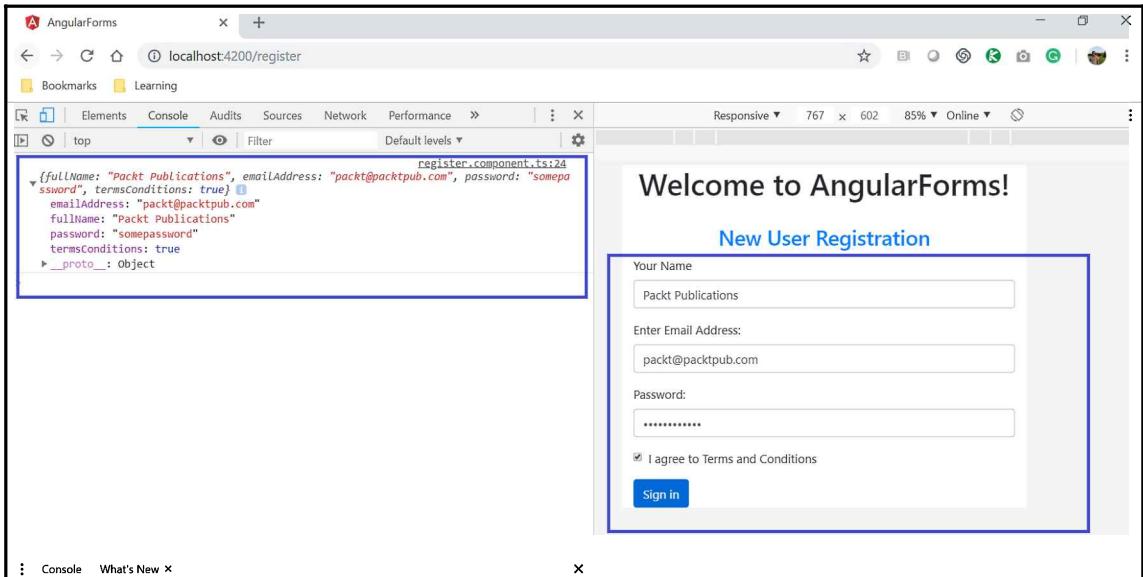
So now, let's update our template file. In the register.component.html file, add the following code:

```
<div>
  <form [formGroup]="registerForm" (ngSubmit)="register()">
    <h3 class="text-center text-primary">New User Registration</h3>
    <div class="form-group">
      <label for="fullName">Your Name</label><br>
      <input type="text" formControlName="fullName" class="form-control">
    </div>
    <div class="form-group">
      <label for="emailAddress">Enter Email Address:</label><br>
      <input type="text" formControlName="emailAddress" class="form-control">
    </div>
    <div class="form-group">
      <label for="password">Password:</label><br>
      <input type="password" formControlName="password" class="form-control">
    </div>
    <div class="form-group">
      <div class="form-check">
        <input class="form-check-input" type="checkbox"
          formControlName="termsConditions" id="defaultCheck1">
        <label class="form-check-label" for="defaultCheck1">
          I agree to Terms and Conditions
        </label>
      </div>
    </div>
    <button type="submit" class="btn btn-primary">Sign in</button>
  </form>
</div>
```

In the preceding code, we are creating a dynamic reactive form. There are many important concepts that we need to understand in the preceding code. We are using the `FormGroup` attribute for model-driven forms. In the template-driven forms, we used `ngForm`. Notice carefully that for every form element, we mention the `FormControlName` attribute, and the value for this attribute has to be exactly the same as was mentioned in the Component class during the `FormControl` instance declaration. Take a pause and read the last few sentences again.

We don't have to mention `ngModel` for elements anymore, since data binding is tightly coupled inside the Component class itself. We have also attached a `ngSubmit` event, which will call the method `register` implemented inside the component to print the form value on the console.

Awesome. That's it. Now serve your application using the `ng serve` command and we should see the output as displayed in the following screenshot:



Congrats on getting your forms up and running using the approaches provided by Angular. We have learned to build forms using template-driven and model-driven approaches. In the next sections, we will learn to extend them by adding validation and custom rules.

Angular form validations

By now, we understand how important and critical forms are to all our applications. Since we will be working on data we receive from users, it's very important to make sure that the data we receive from our users is correct and valid.

For example, when we expect the user to enter an email address, we should not allow spaces or a few special characters in the email address. One more example, if we request the user to enter the phone number, it should not have more than 10 digits (excluding country code of course).

There can be many such custom valid checkpoints that we may want to have in our forms.

In this section, we will continue to use both the login form and our registration form to learn how to add validations in both template-driven forms and model-driven forms.

Template-driven form validation

Bring up our login form that we developed using the template-driven approach.

Remember, that in the template-driven forms, validation is done in the template itself using the HTML attributes.

We can use any of the HTML attributes, such as required, maxlength, minlength, size, email, number, length, and so on, to put validation in forms. We can also make use of HTML pattern attributes to put regular expression checks in our form elements.

There are various classes that we can make use of readily to implement validation in our forms:

- ng-touched: Input controls have been visited
- ng-untouched: Input controls have not been visited
- ng-dirty: Input controls data was changed
- ng-pristine: Input controls data has not been changed/updated
- ng-valid: Input control data is a valid set and makes the form valid
- ng-invalid: Input control data is not valid and hence the form is not valid

In template-driven forms, Angular will automatically track the state of each input element and the state of the form as well. Hence, we can also use the preceding classes in our CSS/SCSS to style/design our error notifications, as follows:

```
input.ng-invalid {  
    border:2px solid red;  
}
```

Alright, now that we know about validations in template-driven forms, it's time to update our login form component and make it jazzy. We are updating the `login.component.html` file by adding validations to the form elements.

```
<div>  
  <form #loginForm="ngForm" (ngSubmit)="login(loginForm.value)">  
    <h3 class="text-center text-primary">Login</h3>  
    <div class="form-group">  
      <label for="username">Username:</label><br>  
      <input type="text" ngModel #username="ngModel" name="username"  
             placeholder="Enter username" required class="form-control">  
      <span class="text-danger" *ngIf="username.touched && !username.valid">  
        enter username </span>  
    </div>  
    <div class="form-group">  
      <label for="password">Password:</label><br>
```

```
<input type="password" [ngModel]="password" name="password"
       required minlength="3" class="form-control">
</div>
<button type="submit" class="btn btn-primary"
[disabled]={!loginForm.valid}>
  Sign in</button>

</form>
</div>
```

Let's take a closer look at the preceding code. We have extended the login form we had created earlier. Notice that for the username form control, we have the HTML attribute `required`, which will be set on the form control. If the user does not enter any value for the field and steps out of the focus of the field, using the `ngIf` condition, we are checking if the field is touched by the user and is if the value is not valid, we are displaying the error message. For the password field, we are setting other HTML attributes, such as `required` and `minlength` validation checks. If the form control data is not valid, we should not enable the form, right? That's what we are doing by adding the `disabled` attribute to the Submit button.

Now let's run the app using the `ng serve` command and we should see the output, as shown in the following screenshot:



For your homework, please try out these use cases in template-driven forms:

- Add minimum and maximum length to the username form element
- Add a new form element and add validation that it should be in an email format

Reactive form, or model-driven form, validations

So far all, the validations we have implemented are only in the template file using the basic HTML attributes. In this section, we will learn to implement the validations in the component using the model-driven forms.

In previous sections, we have learned to create a form using the `FormControl` and `FormGroup` classes in our `Component` class. We will continue to use the same registration form to extend and implement validations now.

We are adding the validation code in our component by adding validations in the `register.component.ts` file. Take a look at the code we will add in the file:

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

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.scss']
})
export class RegisterComponent implements OnInit {
  registerForm = new FormGroup({
    fullName: new FormControl('', [Validators.required,
      Validators.maxLength(15)]), emailAddress:
    new FormControl('', [Validators.pattern('[a-zA-Z]*')]),
    password: new FormControl('', [Validators.required]),
    termsConditions: new FormControl('', [Validators.required])
  });

  constructor() { }

  ngOnInit() {
  }

  register()
  {
    console.log(this.registerForm.value);
  }
}
```

In the preceding code, you will notice that we have imported the required modules, `FormGroup`, `FormControl`, and `Validators` into our Component class. We had already imported and used `FormGroup` and `FormControl`. The `Validators` module is the only additional module that we have imported now. We are passing the validators as options to `FormControl`. For `fullname` we are adding the validators as required and `maxLength`. Note that we can pass multiple validators for each `FormControl`. Similarly, for email address form control, we are passing a validator pattern, which has a regular expression check on it. We have made all the required changes and validations in our component.

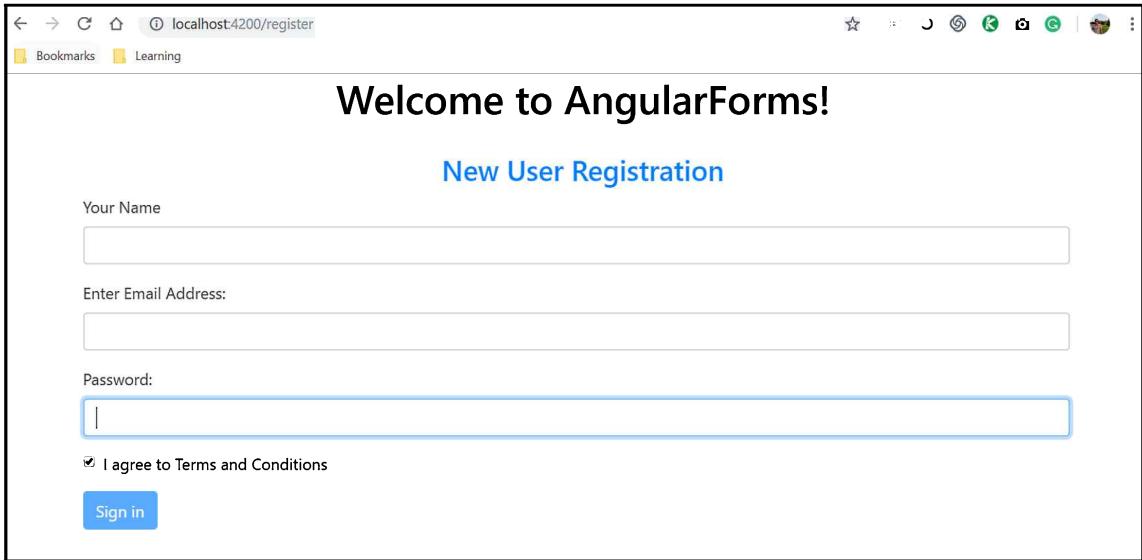
Now it's time to update our template `register.component.html` file:

```
<div>
  <form [formGroup]="registerForm" (ngSubmit)="register()">
    <h3 class="text-center text-primary">New User Registration</h3>
    <div class="form-group">
      <label for="fullName">Your Name</label><br>
      <input type="text" formControlName="fullName" class="form-control">
    </div>
    <div class="form-group">
      <label for="emailAddress">Enter Email Address:</label><br>
      <input type="text" formControlName="emailAddress" class="form-control">
    </div>
    <div class="form-group">
      <label for="password">Password:</label><br>
      <input type="password" formControlName="password" class="form-control">
    </div>
    <div class="form-group">
      <div class="form-check">
        <input class="form-check-input" type="checkbox"
          formControlName="termsConditions" id="defaultCheck1">
        <label class="form-check-label" for="defaultCheck1">
          I agree to Terms and Conditions
        </label>
      </div>
    </div>
    <button type="submit" class="btn btn-primary"
      [disabled]="!registerForm.valid">Sign in</button>
  </form>
</div>
```

The HTML template is the same as the one we had created earlier for our model-driven form. We have added some more functionality to the form. Notice that we have added the `disabled` attribute to the Submit button, which will disable the form if any form elements are empty or invalid.

See, I told you, our template file will just be a placeholder and almost all of the action happens in our Component class.

Now, let's serve the app using the `ng serve` command and we should see the output, as shown in the following screenshot:



If you see the preceding screenshot, jump on your desk. As we have now learned and implemented forms using both template-driven and model-driven approaches.

If you paid attention to the examples covered during the entire chapter, you will also notice that we have created methods to process the form data.

In the next section, we focus exclusively on the same and learn some best practices to process form data.

Submitting form data

So far, we have learned to design and develop our forms in our applications. In this section, we will take things to downstream systems, which is to capture the data and process it.

Angular generates a form model in both approaches, be it template-driven forms or reactive forms. The form model holds the data of the form elements and the state of the form elements.

In the previous sections, where we have implemented our forms, we have created a method to call on `ngSubmit`.

For our template-driven login form, we added the following code to our `login.component.ts` file:

```
login(loginForm)
{
  console.log(loginForm);
  console.log(loginForm.username);
}
```

We are passing the entire form object to the `login` method. Now the `loginForm` object will have all the details of the form controls, as well as the states.

In our registration form, which is generated using a model-driven approach, we have used the instance of `formGroup` that we created in our `Component` class `register.component.ts` file.

The following is the code we have added for capturing and processing the data:

```
register()
{
  console.log(this.registerForm.value);
}
```

If you notice, for reactive forms, we do not have to pass any form data, since we had created the `registerForm` instance of `FormGroup`, so it's available in our class using the `this` operator.

Once we have captured the data provided to the user, based on the application requirements, we can now implement our custom logic inside the component.

Some of the common activities we do once we capture data are the following:

- Securing data to make sure that we do not allow junk data into our system.
- Processing/enhancing the data, such as converting the password into an encrypted value.
- Checking for any automated bots processing our applications.
- Making HTTP calls to backend services using Angular services. We have an entire chapter dedicated to this particular topic: Chapter 12, *Integrating Backend Data Services*.

That concludes this chapter on Angular forms. We have covered a lot of ground and I am sure at this point you will be excited to create your own forms, write custom validations, and process the captured data.

Summary

Forms are the very heart and soul of any good application. We started by learning the awesome classes and utilities provided by the Bootstrap library. We explored in detail the `form-group` and `form-control` classes. We learned and implemented various helper and additional attributes that we can use to make our form look and behave even better.

We deep-dived into Angular forms by learning the two approaches Angular provides, namely template-driven forms and model-driven forms.

We learned in detail about each of the approaches, their pros and cons, and above all created our login and registration forms using each of the approaches. We also explored the various type of validations that we have used with template-driven forms and reactive forms.

Last, but not least, we learned about processing the form data that we receive from the forms. It's time to get wings and create your awesome forms.

While developing complex applications with multiple developers working on the same application, things can get out of hand. Luckily, Angular supports dependency injections and services, which enables us to create reusable services and define interface classes. We can define new data types and make sure all team members are pushing code without breaking each other's functionality. And how exactly will we achieve that? That's covered in the next chapter. Keep reading!

11

Dependency Injection and Services

In this chapter, we're going to take a look at **dependency injection (DI)**. While DI is not something that you have to program directly in Angular (since Angular takes care of all the DI plumbing for us), it still is something that is very useful to understand. This is because Angular makes heavy use of DI when managing its services, as well as any custom services that you are likely to write as you create your Angular applications.

We'll be looking at one of Angular's most important built-in services, its HTTP service, in the next chapter, *Chapter 12, Integrating Backend Data Services*. Without the HTTP service, our applications would be super boring, since they would be unable to send data to, or receive data from, an outside source (including our own backend APIs). And so this chapter will serve us well in terms of helping us gain an understanding of how Angular injects services such as its HTTP service into our applications for us to use. Moreover, this topic is a perfect segue into the next chapter.

Here is a list of the topics that we're going to cover in this chapter:

- What is DI?
- What problem does it solve?
- Additional advantages of using DI
- Revealing the magic that Angular uses to make it all work
- How we can guard against code minification (and why we need to)

By the end of this chapter, you'll have a solid grasp on what this often misunderstood software design pattern is, and, more importantly, how it works. Dare I say, you may even start to feel more technically advanced than most of your peers. Many developers sometimes struggle to even define DI—because it does take some effort to wrap your mind around it.

Without further ado, let's get started and discover what DI is all about by *injecting* some more software design knowledge into our heads.

What is DI?

Cutting to the chase, DI is a specific case of the **Inversion of Control (IoC)** design pattern.

In order to understand this high-level definition of DI, or even IoC, we first need to quickly define design patterns. Design patterns are reusable solutions to common problems in software design. There are dozens of software design patterns, and they are typically categorized into the following three high-level categories:

- Creational patterns
- Structural patterns
- Behavioral patterns

In our case, and in the interest of brevity, we can safely ignore the creational and structural categories of design patterns, since DI is a type of behavioural design pattern, and just before we take a look at the IoC design pattern (I know, there are a lot of definitions and concepts we need to know as prerequisites to truly appreciate what DI is), let's describe what behavioral design patterns are.

Simply put, behavioral design patterns concern themselves with how objects communicate with one another. One such pattern is known as the observer design pattern, which basically sets forth a way of how an object notifies its dependent objects of when its state changes.

Another behavioral design pattern is known as the publish-subscribe design pattern, which is a messaging pattern similar to the observer pattern, but a little fancier. Yet another behavioral design pattern is the template method. The purpose of this design pattern is to defer the exact implementation of an algorithm to a subclass. The overall idea behind all of these design patterns is the way they communicate (that is, message) with one another.

Armed with a definition of what the template method is, we're getting much closer to what DI is all about, but just before we do, there is one last remaining design pattern that we need to define. You guessed it—the IoC design pattern. Remember, DI is a special case of the IoC pattern, and so we really do need to take a quick look at what it is.

IoC flips the typical flow of procedural, or imperative, code on its head. Instead of having a custom object's code control the flow of the program by instantiating another object and then calling one or more of the newly instantiated object's methods, it defers the instantiation to a framework—yes, a framework, not just another object—to do that instead. This will all make sense in a few moments. As an interesting tidbit, this is sometimes jokingly referred to as *Don't call us, we'll call you*.

We're going to look at an example of this shortly so that it all makes sense. However, I need to define what I mean by a framework doing the instantiation of dependent objects. Don't you just love all the terms and concepts we need to know? (grinning). This framework is typically referred to as an IoC container. These containers are smart enough to be able to examine the custom code, figure out what other objects it depends on, instantiate those objects, and pass them into the custom object's constructor. This is opposed to the traditional way of having the instantiation of the object's dependencies happen within the custom object itself. Instead, the IoC container performs these duties for it. In a moment, I will tie this back to Angular and give you a couple of very important advantages that the IoC pattern provides, but we'll discuss it from the DI perspective—finally!

OK. Let's try to tie this all together and provide an example scenario, or use case. The Angular framework provides the functionality of an IoC container—among all the other things that it provides. Since Angular is a modular framework and encapsulates most of what it does in segregated services, it would make sense that its IoC capabilities are also encapsulated in one of its services—and, in fact, this is the case.

The Angular service responsible for DI is its injector service—aptly named because it injects your custom class's dependencies into your class's constructor after instantiating them. Not only that, but it calls the custom method for you—back to what I had previously mentioned, *Don't call us, we'll call you*. All we need to do is to list the names of the dependencies in the constructor's signature for our custom class.

From this point forward, I will not mention IoC, because we're talking about DI—again, this is technically not IoC, but rather a special case of it. I only mention this because many developers use the terms IoC and DI synonymously.

So, let's ask a couple of questions: Since DI is a design pattern, and design patterns solve common software design problems, what problems does DI solve? What are the advantages of DI? These are excellent questions, and I believe I can answer them in one fell swoop in the following two paragraphs.

One problem that even object-oriented code has had for a very long time is that a class that depends on other classes (which is the whole point of object orientation—since we don't want one class to do all of the work) had the code to instantiate these dependencies within itself, and, as a result, contained at least some of the logic intertwined within it as well. This is known as code that is tightly coupled. There are two problems with tightly coupled code: firstly, the implantation logic is typically encapsulated within the class—which is something we don't want. We don't want one object to know the inner workings of other objects. For instance—if we wanted to change the implementation of an algorithm in a dependent class, we'd likely also have to change our code in the class that calls it. Another problem that stems from this is that this code is difficult to test. The more tightly coupled our classes are, the harder it is to run our unit tests on them—and this problem has been around for as long as unit testing has.

OK. So how does DI solve these issues? We'll get to a specific use case to make everything clearer in our minds, but let's first describe a couple of the advantages that DI gives us. The first advantage of the principle of DI is that it forces us to write decoupled code. We do this by having the classes that we depend on (for their abstracted implementations) implement interfaces, and we do this so that all our calling class needs to do is to call the interface methods on these objects—not caring about the implementation details behind the underlying class methods. When we write code in this manner, we can swap out the class we depend on, which has a specific implementation, for another class that has another implementation—all without changing any of our calling code (since our code calls the interface methods that these classes implement). This is also sometimes referred to as coding by the interface. Here's another interesting tidbit: this technique is also used in a style of programming known as aspect-oriented programming, or AOP for short.

A very useful thing that we get for free by adhering to the DI design principal, is that we can very easily test our code—as opposed to not being able to test our code easily, or at all, as is the case when we have tightly coupled code. How do we do this? By writing stubs and/or mock classes—which also implement these very same interfaces that our calling class calls.

As a side note, there is an important difference between stubs and mocks. Stubs are dumb classes, which often just return a simple value (often hardcoded). Mock objects, on the other hand, typically have full implementations so that things such as edge cases can be tested—as well as to conduct database operations, or make RESTful API calls. Mocks can be used to do whatever your tests call for. So, stubs are dumb, while mocks are smart. What they have in common, however, is that they help us unit test our calling class's code by having the same object messaging pattern (that is, their methods are called via interfaces).

Whew! We're done with the theory part! Have you fallen asleep or are you still with me? Ah, you're awake—OK, good. With all that theory now out of the way, let's take a look at an example use case of where DI can be used for all the reasons mentioned previously—just so we can cement these concepts into our heads.

Let's assume that we're building an e-commerce application for an online store where we sell our homemade beer. Our application will need to have a shopping cart, and we'll also have to have at least one merchant account (which is a conduit, known as a payment processing gateway, so we can charge our customers' credit cards). In this hypothetical scenario, we have two merchant accounts—maybe because we'd like to keep one as a spare in case the primary merchant account increases their discount rates (that is, fees), thereby lowering our profits—but the point is, we have two merchant accounts.

When it comes to implementing the shopping cart, we want to be able to swap out one merchant account for another one—if we need to—without changing the code in our shopping cart class. The reason we don't want to change any code is that we may accidentally introduce bugs in our application (the online store), and this just wouldn't look good to customers. You may be saying—*Hey, I test my code—so bugs are fleshed out*—and if you said that, you fell right into the next benefit of using DI for our application, which is that we can easily test our application by writing test classes—remember our stubs and mocks? Yes—we write stubs and mocks so we can test our code. And again, thanks to DI, we don't have to change our shopping cart class to do so. We have our stubs and mocks implement the interfaces. And we would wrap the bank's APIs (that is, our merchant account classes—written by a third party) in a custom class that implements our interface—so that all these classes (that is, our stubs, mocks, and wrapped real bank objects) can be called in the exact same way.

Cool. So, as a bonus, let's take a quick look at how Angular knows what our class needs, and how it can call our class's constructor method for us. Well, it's not magic, but it is ingenious. However, Angular does need a little upfront help from us. When we create custom classes, for our application, we typically wrap them up as Angular services (we'll take a look at services in the next chapter, Chapter 12, *Integrating Backend Data Services*). Angular requests that we register these services with it, and you'll see why we need to do this in a moment.

Angular's injector service scans our code, specifically, our class's constructor signature, and figures out its parameters. Because our parameters are services that we need for our class, it knows that the parameters are services. It then matches the text of the service name against the manifest of its own services, as well as any custom services that we wrote ourselves, and when a match is found, it instantiates that service object. The reason it can do this is that it knows its own services, and it knows what services we wrote because we had to register them with Angular.

The next thing Angular does, once it has these instantiated service objects, is to call our class's constructor, passing the objects in as the arguments. That is the injection process that Angular's injector service does. One more time, say it with me: *Don't call us, we'll call you.* And just like that, the magic behind what Angular does has been explained away. Still, it's very cool and we should tip our hat to the Angular development team.

Generating services and interfaces

Now that we have learned about DI and design patterns, in this section, we will learn to create our services. Angular CLI provides us with the fastest and easiest way to generate services inside our project. We will create a sample project called `LearningDIServices` by running the following command:

```
ng new LearningDIServices
```

We are creating a new Angular project using the `ng` command, and we name the project `LearningDIServices`. Upon successful execution of the command, we should see the output shown in the following screenshot:

```
$ ng new LearningDIServices
CREATE  learningDIServices/angular.json (3876 bytes)
CREATE  learningDIServices/package.json (1318 bytes)
CREATE  learningDIServices/README.md (1035 bytes)
CREATE  learningDIServices/tsconfig.json (435 bytes)
CREATE  learningDIServices/tslint.json (2837 bytes)
CREATE  learningDIServices/.editorconfig (246 bytes)
CREATE  learningDIServices/.gitignore (576 bytes)
CREATE  learningDIServices/src/favicon.ico (5430 bytes)
CREATE  learningDIServices/src/index.html (305 bytes)
CREATE  learningDIServices/src/main.ts (372 bytes)
CREATE  learningDIServices/src/polyfills.ts (3234 bytes)
CREATE  learningDIServices/src/test.ts (642 bytes)
CREATE  learningDIServices/src/styles.css (80 bytes)
CREATE  learningDIServices/src/browserslist (388 bytes)
CREATE  learningDIServices/src/karma.conf.js (980 bytes)
CREATE  learningDIServices/src/tsconfig.app.json (166 bytes)
CREATE  learningDIServices/src/tsconfig.spec.json (256 bytes)
CREATE  learningDIServices/src/tslint.json (314 bytes)
CREATE  learningDIServices/src/assets/.gitkeep (0 bytes)
CREATE  learningDIServices/src/environments/environment.prod.ts (51 bytes)
```

Now that we have our project directory created, using the Angular CLI, we will generate a few services and interfaces. We will create a service called `Photos`. Run the following command and we should see the service added to our project directory:

```
ng generate service photos
```

Upon successful execution, we should see the output shown in the following screenshot:

```
PS D:\book_2\book\final_code\chapter11_dependency_services\learningDIServices> ng generate service photos
CREATE src/app/photos.service.spec.ts (333 bytes)
CREATE src/app/photos.service.ts (135 bytes)
PS D:\book_2\book\final_code\chapter11_dependency_services\learningDIServices> []
```

We can see that there are two new files generated. One is the service file, and the other is the spec file, which is for writing the tests for the service. Let's take a closer look at the files containing autogenerated code for the `photo.service.ts` file:

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

@Injectable({
  providedIn: 'root'
})

export class PhotosService {
  constructor() { }
}
```

In the preceding code, we can see that the `Injectable` class needs to be imported from the `angular/core` library. The `Injectable` class allows us to make the service inject in various component classes so that we can reuse the methods. Using the `injectable` decorator, we are explicitly mentioning that the service needs to be injected in the root. And finally, we are exporting our `PhotosService` class, which will contain the `constructor` method and other methods that we will create, specific to our application.



Unlike the Angular components, there is no need to update `app.module.ts` file with an entry of the service.

In previous sections, we learned about the overview of interfaces. Now, let's quickly learn how to generate use of the interfaces in our applications. Using Angular CLI, we can also quickly create interfaces:

```
ng generate interface photo
```

In the preceding command, we have generated an interface named `photo` and, once the preceding command is executed successfully, we should see the following output:

```
PS D:\book_2\book\final_code\chapter11_dependency_services\learningDIServices> ng generate interface photo
CREATE src/app/photo.ts (27 bytes)
PS D:\book_2\book\final_code\chapter11_dependency_services\learningDIServices> █
```

Let's take a closer look at the interface files that are generated. The following is the default code generated:

```
export interface Photo {  
}
```

We can see it's empty intentionally. Since interfaces are used to define the entity or model classes, each interface created in the application will be unique and specific to each application. Now, if we want to create an interface for our photos, we will have to define it as follows:

```
export interface Photo {  
  photoId: number;  
  photoURL: string;  
  photoOwner: string;  
  isActive: boolean;  
}
```

In the preceding sample code, we have created an interface for photos with some properties and their data types. This will allow us to create strictly typed objects for photos.

In this section, we learned to create Angular services and interfaces. Even if some of the concepts are not very clear, do not worry, my friend. We have an entire chapter dedicated to showing you how to generate and implement services in our applications. In the next chapter, we will learn how to implement and use them, and also integrate them into our components.

Guarding against code minification

There is one last thing that I would like to cover really quickly, and that is code minification and how we can guard against it. Code minification is the process of compressing our code by removing the whitespace, as well as replacing variable names with very short symbols. This is done when we compile our Angular application so that it becomes a smaller package that our users have to download (once we've deployed our application) to retrieve our application. But this does present a problem for us. It can wreck our day by changing the parameter names and then Angular can no longer match the names against the service manifest. Fortunately, there is an easy solution. If we add single quotes around our parameter names, we protect our code against code minification. How? Well, putting quotes around the service names turns them into literal strings, and the minification process does not compress or change strings—it leaves them intact. This is because literal strings have a meaning outside of syntax, and is not code. Minification just minifies code (that is, variable and function names, and whitespace). That's all you have to know about protecting your code from code minification.

Summary

You should now feel comfortable with what DI is, and what problem it solves. You should also be able to list a few of the advantages—thus being able to explain why DI is a good principle for us to follow in designing our applications. And you should also be comfortable in explaining away the seemingly magical feats that Angular performs in making everything work out of the box. And finally, you should also now know how to guard your DI code against code minification.

Armed with this DI knowledge, we can now continue our journey into discovering one of Angular's most useful services, its HTTP service, in Chapter 12, *Integrating Backend Data Services*. Once you complete the next chapter, you will be ready to write code to integrate your Angular application with virtually any RESTful API-compliant application and/or service that your application is authorized to talk to. That should sound exciting to you! If it does, turn the page and continue your journey to Angular enlightenment.

12

Integrating Backend Data Services

Welcome to Chapter 12! This is definitely my favorite chapter, as we will be building many end-to-end use cases for our application.

A gentle warning—this chapter is dense—it is packed with a ton of information. You may have to read it at a slower pace and spend more time at the keyboard than you have with the previous chapters, but, I must say, it is well worth the effort.

Here is a great way to look at the overall progression of this book:

- Everything we've looked at so far, including the two most recent chapters ([Chapter 10, Working with Forms](#), and [Chapter 11, Dependency Injection and Services](#)), has laid the foundation for this chapter. With that knowledge under our belts, we are now ready to put it all together in order to create our application. So, in essence, this chapter also serves the purpose of reviewing many of the topics we covered in previous chapters.
- This chapter is the pivotal turning point for us because we will take everything we've learned thus far to build 95% of our application in this single chapter. This is a lot of material for one chapter, but we have spent a good amount of time going over all the aspects of Angular that we will need to build our application, so we're going to breeze through it. There is some new and slightly off-topic material too—learning how to build backend APIs—which is less important than the Angular material. However, we need to have an API, so I selected a set of technologies that are simple enough to quickly get up to speed with. We are also going over this to help you get your mind around the technologies we will be using to construct the APIs.
- In the chapters that follow this one, we will add a couple of things to our app (such as route guards and custom form validation) and we will learn how to test, debug, secure, and deploy our application.

So, from that perspective, we're good to go. Many sections in this chapter are bonus material that I deem important to learn about because I want you to succeed, not only as an Angular developer, but as a web developer in general. This will help you to enhance your skills, and hands-on examples are sure to augment your technical knowledge as a web developer.

We will cover the following topics:

- ListingApp – an overview
- Fundamental concepts for Angular applications
- ListingApp – technical requirements
- Building APIs for our application
- The Google Firestore database
- Angular HttpClient
- Integrating backend services

We've spent a lot of time in this book discussing a myriad of things—mostly Angular-centric (such as components, routing, flex-layout, NG Bootstrap, Angular Material, and working with forms), and a few things that were standalone (such as wire-framing, ES6, TypeScript, and Bootstrap). It is important to possess all that knowledge, of course, but we haven't yet integrated live data to bring our Angular application to life. However, as you can see from the previous bullet-point list, this is about to change. This is where developing in Angular starts to get fun, and also much more practical, since an application that does not create and consume data is not much of an application at all.

OK. Let's get right into it by starting with learning about some of the fundamental concepts that form the foundation of any application. Then, we'll take a look at the steps involved in building our ListingApp.

ListingApp – an overview

In this chapter, we will be building our ListingApp application. In this section, we will cover the functional requirement list. Our overall application plan can be broken down into three main sections:

- **UI layer:** The UI aspects involve designing or building the forms, displaying data, routing, and validations.

- **Services or middleware layer:** We will learn how to write shared services, which will be responsible for backend integrations with APIs and databases.
- **Database or fake API setup:** We will learn how to set up fake APIs using JSON Server and we will also learn how to create our NoSQL database using Firestore.

Here's the complete list of functional use cases we will be building as part of the learning experience in this chapter:

- Display all the listings
- View listings by ID
- Add a new listing
- Edit a listing
- Delete a listing
- Add comments
- Update comments
- Delete comments
- Edit comments

All the use cases listed will require us to implement HTTP calls. For some, we will need to make POST, GET, and PUT HTTP calls.

Before we proceed any further, now is a good time to bring back all the learning and functionality we have implemented throughout the course of this book. We will need to recollect how we designed and developed our forms, how we captured the form data, how we displayed the data in the component templates, how we implemented routing with parameters, and how we call methods implemented in services inside components.

We have a lot of work to do, and a load of fun awaits us, so let's get started!

Fundamental concepts for Angular applications

We are going to learn and build a lot of interesting things in this chapter, but before we start doing that, we should learn about a number of fundamental concepts, including strongly typed language concepts, Angular models, observables, NoSQL databases, and CRUD operations in general.

Strongly typed languages

A strongly typed programming language refers to the fact that each of the data types is predefined and tightly coupled with the variables. Take a look at the following variable that's been defined:

```
int age = 10;
```

We are declaring a variable and explicitly mentioned the fact that the type of variable is an integer, which makes it very obvious that the variable cannot hold any other data type except an integer. If we try to provide any value that is not an integer, TypeScript will throw an error. TypeScript is also a strongly typed language and, since we write our Angular applications in TypeScript, we can conclude that Angular applications follow strongly typed formats.

TypeScript interfaces

In this section, we will learn how to create our own data types in TypeScript, which we can use in our Angular applications.

Angular models are a way to create complex data structures by clubbing multiple data types into an object and defining a new object, which can then be used as a data type in itself. It's Angular's way of ensuring that complex data objects adhere to certain predefined data specs.

The TypeScript language provides interfaces that also serve the same purpose. We can also make use of ES6 classes to define our data structures. We can extend the programming syntax to create our custom data types. Let's demonstrate this by creating a sample model. We are going to create a model and call it `Listing`, which will have the following properties:

```
export class Listing {  
    id: number;  
    userId: number;  
    title: string;  
    status: string;  
    price: number;  
    active: boolean;  
}
```

We have created an Angular model, which is a class with properties such as `id`, `userId`, `title`, `status`, `price`, and `active`. We can now use this model as a data type in our application. We can import this class into all of our components and services to make sure our data map adheres to the Listing data spec.

We will use the previously defined model throughout this chapter while building our application.

Observables

Most traditional applications work on the request and response architecture, which means our application client will make a request for data to the server, and, in return, the server will give us a response. While the server is returning the response, our application goes into wait mode until all the responses are received, which obviously makes applications slow.

This architecture has multiple drawbacks. First, the application waits for a response, which creates delays in applications. Second, there is no way we can handle multiple pieces of data coming in over a period of time. Third, since our application waits until it gets the response, which makes synchronous calls, we cannot execute asynchronous programming. And finally, event handling becomes a nightmare for developers. So, how do we address the preceding issues? The answer is by using observables.

Observables are a type of array that returns data over a period of time asynchronously. Angular uses a third-party library called **Reactive Extensions (RxJS)** and has observables implemented inside the framework mainly for event handling, tree shaking, and so on. We can also easily import, create, and subscribe to custom observables.

NoSQL databases concept

In this section, we are going to learn about NoSQL databases. Really? NoSQL? Are we not going to use a database to store our critical data? Of course we are going to use a database to store our data; however, it won't be a traditional relational database, which has a strict predefined schema and columns that have a standard data type. With NoSQL databases, everything is document-oriented and we can store data in one place without worrying about the data types. NoSQL databases hold collections of documents.

We can still perform database activities such as the following:

- Creating a document
- Inserting a document
- Editing an existing document
- Deleting a document

We can also perform a lot of advanced functionality, such as indexing and authentication. There are a lot of open source as well as commercial solutions that provide NoSQL databases. Here's a quick list of some of the NoSQL database providers:

- MongoDB
- Redis
- RavenDB
- Firestore
- MemcacheDB

During the course of developing our application in this chapter, we will implement Firestore as our backend system. In the next section, we will learn about some of the important tasks we will undertake involving these databases.

CRUD operations – overview

Whenever we consider using a database as a backend storage system for applications, the primary goal is to be able to add, retrieve, search, or modify the data, which are more commonly known as CRUD operations.

CRUD stands for Create, Read, Update, and Delete in computer programming, and these terms are outlined as follows:

- **Create:** Create or add new data to the database. We would usually be running an INSERT query in the database. This is associated with the HTTP POST method.
- **Read:** Read or retrieve data based on a filter or search criteria. We will run the SELECT query in the database to do this. This is associated with the HTTP GET method.
- **Update:** Update or edit an existing record in the database. We will use the UPDATE query in the database. This is associated with the HTTP PUT method.
- **Delete:** Delete an existing record in the database. We can either use the DELETE query to delete a record, or just set a column indicating that the record has been deleted using the UPDATE query. This is associated with the DELETE method.

In the upcoming sections, we'll use these concepts to build our ListingApp functionality and the technical requirements of our application.

ListingApp – technical requirements

Any good dynamic application will require us to work with APIs and we will need to store the data in a database. This section covers two very important technical aspects required for building any dynamic application—JSON APIs and a dynamic database. We will be making use of the JSON server and for the database, we will use Google's Firestore database.

Building APIs for ListingApp

During the development cycle of any project, as a frontend developer, we will need to work with APIs and integrate them into our application. We will need to define and agree upon the JSON contracts that we expect from our APIs. In this section, we will learn about the various options we have to generate the APIs that we can use while the backend developers are still working on developing actual APIs. When we have fake APIs available, developers can work independently.

There are various tools and libraries (available for free) that we can use to work with fake APIs. We will be using the JSON server library to serve our APIs. So, let's begin by taking the following steps:

1. To install the json-server library, run the following command in the command-line interface:

```
npm i json-server --save
```

When the command has run successfully, you should see the following output:

```
D:\book_2\book\chapter12_backend\ListingApp>npm i json-server
npm [WARN] optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\fsevents):
npm [WARN] notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ json-server@0.14.2
added 95 packages from 36 contributors and audited 42039 packages in 52.302s
found 0 vulnerabilities
```

2. Now that we have installed our json-server library, it's time to create our APIs and JSON structure. In our project directory, we will create a new folder called APIs and create a new file named `data.json`, which will hold our JSON data. Take a look at the folder structure once you have created the folder and files:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "CHAPTER12_BACKEND". A yellow box highlights the "apis" folder, which contains "data.json" and "listings.json".
- Code Editor:** Displays the content of "data.json". The JSON structure includes "listings", "users", and "cities" arrays.
- Terminal:** Shows the command line interface with the following text:

```
Microsoft Windows [Version 10.0.17131.523]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\book_2\book\chapter12_backend>cd ..
```

```
{
  "listings": [
    { "id": 1, "title": "Sunset in New York", "price": "190", "status": "Active" },
    { "id": 2, "title": "Dawn at Miami", "price": "150", "status": "Active" },
    { "id": 3, "title": "Evening in California", "price": "70", "status": "Inactive" }
  ],
  "users": [
    { "id": 1, "username": "andrew", "userEmail": "andrew@localhost.com" },
    { "id": 2, "username": "stacy", "userEmail": "stacy@localhost.com" },
    { "id": 3, "username": "linda", "userEmail": "linda@localhost.com" },
    { "id": 4, "username": "shane", "userEmail": "shane@localhost.com" }
  ],
  "cities": [
    { "id": 1, "name": "New York" },
    { "id": 1, "name": "California" },
    { "id": 1, "name": "Miami" }
  ]
}
```

3. Since we have created two JSON files, it's time to add some JSON data to the files for listings, as well as users. Open the `listings.json` file by adding the following data to it:

```
{
  "listings": [
    { "id": 1, "title": "Sunset in New York", "price": "190",
      "status": "Active" },
    { "id": 2, "title": "Dawn at Miami", "price": "150",
      "status": "Active" },
    { "id": 3, "title": "Evening in California", "price": "70",
      "status": "Inactive" }
  ],
  "users": [
    { "id": 1, "username": "andrew",
      "userEmail": "andrew@localhost.com" },
    { "id": 2, "username": "stacy",
      "userEmail": "stacy@localhost.com" },
    { "id": 3, "username": "linda",
      "userEmail": "linda@localhost.com" },
    { "id": 4, "username": "shane",
      "userEmail": "shane@localhost.com" }
  ]
}
```

```
        { "id": 3, "username": "linda",
          "userEmail": "linda@localhost.com" },
        { "id": 4, "username": "shane",
          "userEmail": "shane@localhost.com" }
      ],
      "cities": [
        { "id":1, "name": "New York" },
        { "id":1, "name": "California" },
        { "id":1, "name": "Miami" }
      ]
    }
```

We are creating dummy data of JSON arrays for listings, users, and cities. Technically, in a real application scenario, this data would be retrieved from the database at runtime.

4. To start serving the fake APIs with data, we need to start and initialize the JSON file. We will navigate to the API folder where we have created our `data.json` file and run the following command:

```
json-server --watch data.json
```

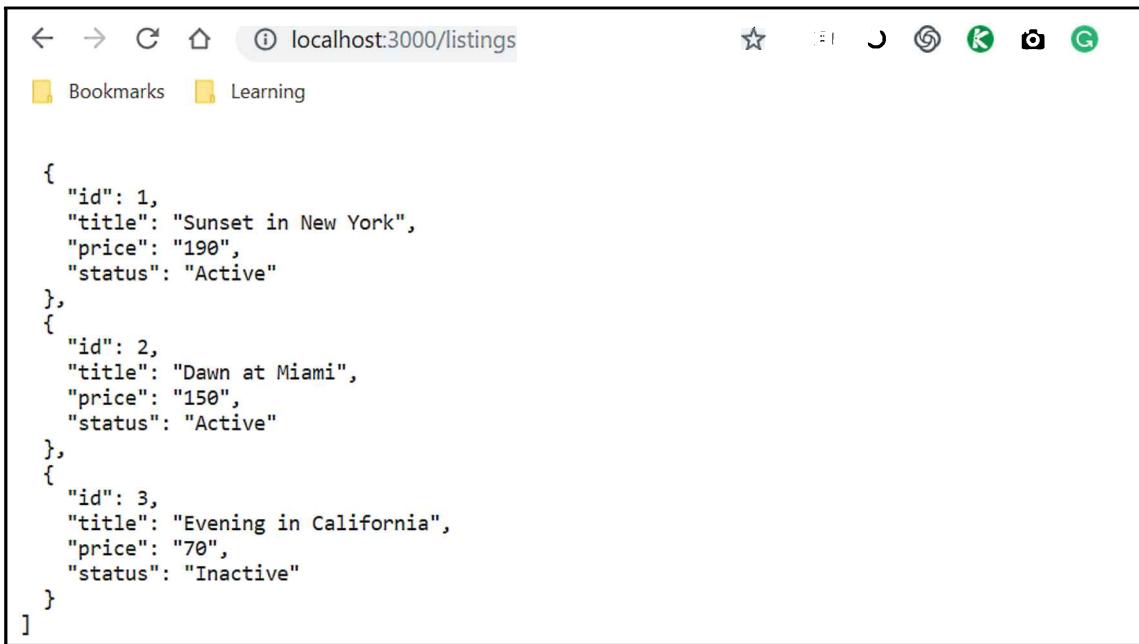
5. When we have run the command successfully, we should see the following output:

The screenshot shows a Windows Command Prompt window titled 'cmd.exe'. The command entered is 'D:\book_2\book\chapter12_backend>cd ListingApp\src\app\apis' followed by 'D:\book_2\book\chapter12_backend\ListingApp\src\app\apis>json-server --watch data.json'. The output shows a JSON object with a 'Resources' key containing three URLs: 'http://localhost:3000/listings', 'http://localhost:3000/users', and 'http://localhost:3000/cities'. Below this, there is a link 'Home' with the URL 'http://localhost:3000'.

```
D:\book_2\book\chapter12_backend>cd ListingApp\src\app\apis
D:\book_2\book\chapter12_backend\ListingApp\src\app\apis>json-server --watch data.json
{
  "Resources": [
    "http://localhost:3000/listings",
    "http://localhost:3000/users",
    "http://localhost:3000/cities"
  ]
}
Home
http://localhost:3000
```

Notice that, under **Resources**, we can see the fake APIs that are listed; that is, `http://localhost:3000/listings`.

6. Try launching the URL in the browser. You should see the JSON data displayed for **listings**, **users**, and **cities**. The output is displayed in the following screenshot:



A screenshot of a web browser window. The address bar shows the URL `localhost:3000/listings`. The page content displays a JSON array of three objects, each representing a listing with fields: id, title, price, and status.

```
{
  "id": 1,
  "title": "Sunset in New York",
  "price": "190",
  "status": "Active"
},
{
  "id": 2,
  "title": "Dawn at Miami",
  "price": "150",
  "status": "Active"
},
{
  "id": 3,
  "title": "Evening in California",
  "price": "70",
  "status": "Inactive"
}
```

Awesome! We can now use these APIs in our HTTP calls. We will have to wait for just one more section before we jump right into learning about HTTP functionality. For our friends who are full stack developers and know how to set up databases, the next section is certainly for you. We will learn about setting up our Firestore database, which will store our data. Later, we will use this to implement our application.

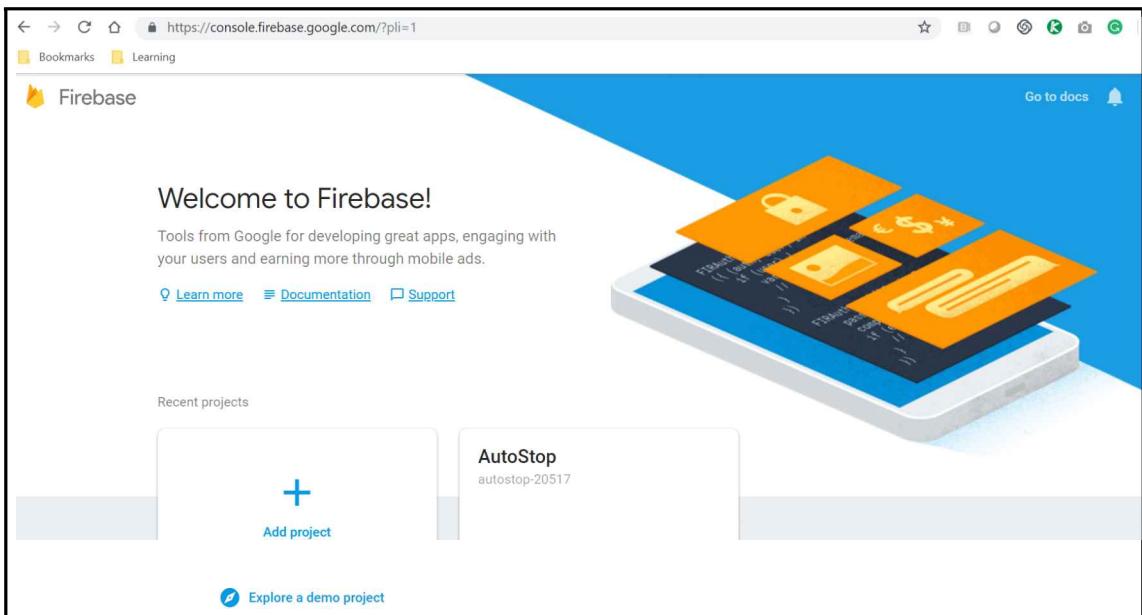
Google Firestore database

The Google Firestore database is part of Google Cloud Platform. The official website of Google Cloud describes it as follows:

Cloud Firestore is a fast, fully managed, serverless, cloud-native NoSQL document database that simplifies storing, syncing, and querying data for your mobile, web, and IoT apps on a global scale. Reference: <https://cloud.google.com/firestore/>

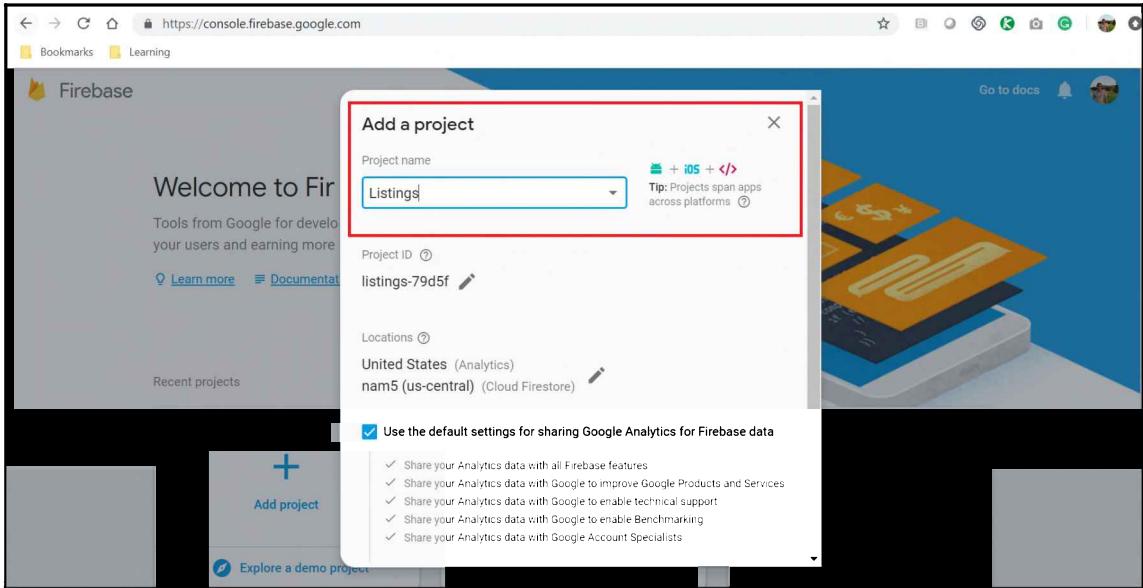
Firebase is a database as a service offered by Google and offers an easy-to-use NoSQL document database. Since Firestore is also coming from the makers of Angular, it's natural that there are libraries that support easy integrations between the two. In this section, we will learn how to set up the Firestore database. So, let's begin:

1. We will need to log in to our Firebase application using our credentials. Upon successful login, we should see the welcome screen, as displayed in the following screenshot:



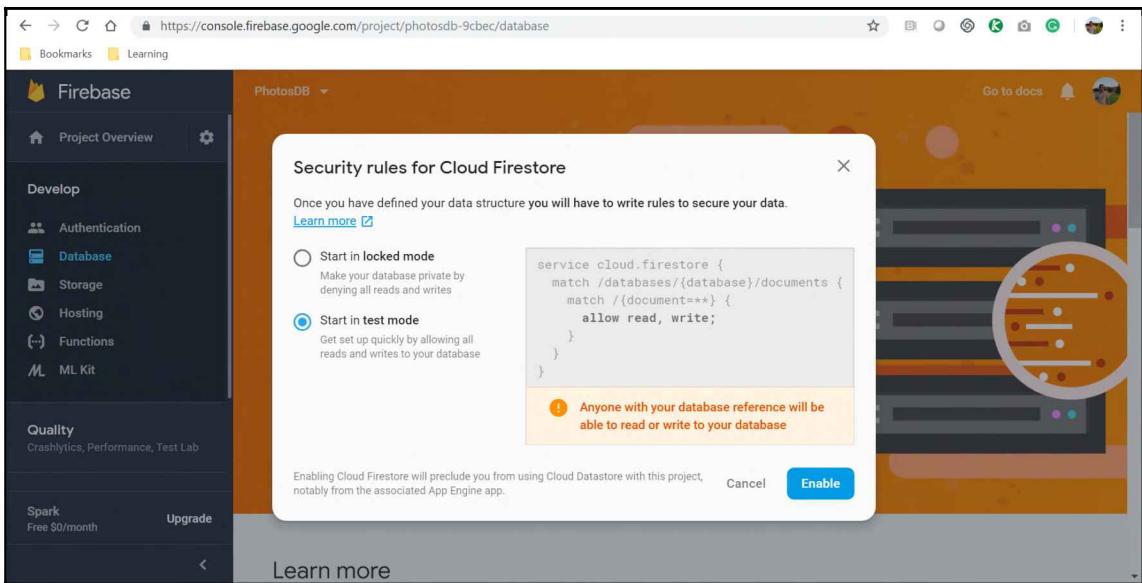
The home page will list all the projects we created in the Firebase application, and you will also notice a big **Add project** link.

2. Now, let's create a new project for our application by clicking on the **Add project** link. We will be prompted by a modal window where we need to enter a **Project name** for our project, as shown in the following screenshot:



Here, we will enter **Listings** as our project name. Once our project has been provisioned, we will be taken to the newly created project page.

3. We now click on **Databases** in the sidebar menu. We will be prompted to select the mode in which we will initialize our database. We will select the test mode for our testing, and once we have performed the implementation, we will switch the security mode:



As shown in the preceding screenshot, we are using a database in test mode, which will allow us to read or write documents easily.



Do not forget to change the settings of the database if you want to use the database in a production environment.

4. We'll now move on to create our `comments` collection. We'll add a unique identifier called `commentId`. In addition, we are adding three fields as a schema for the documents that will be stored in the collections, as shown here:

The screenshot shows the Firebase Database interface. On the left, the sidebar includes 'Project Overview', 'Authentication', 'Database' (which is selected), 'Storage', 'Hosting', 'Functions', 'ML Kit', 'Crashlytics', and 'Spark'. The main area shows a 'Listings' collection with a 'comments' subcollection. A specific document, 'Ydfgpi8s6SSU80Lhvsh', is selected. This document has fields: 'commentId' (with a value of 'Ydfgpi8s6SSU80Lhvsh'), 'commentTitle' (empty), 'createdOn' (empty), and 'userId' (empty). A red box highlights the 'comments' subcollection path in the left navigation tree.

Since Firestore is a NoSQL document database, the schema is not restricted by any data types. We can now perform CRUD operations, such as adding a new document, editing, or even deleting documents, in the Firestore database.

In the last two sections, we have learned about creating fake APIs using the JSON Server, and we have also created a NoSQL document database using Firestore. Now that we have reached a stage where we have learned about all the fundamental concepts needed to start implementing the end-to-end functionality of our ListingApp, let's jump into the HTTP world!

Angular HttpClient

In this section, we will learn about the most important aspect of Angular—`HttpClient`. Using the `HttpClient` interface, we can perform HTTP request and response calls. In the previous chapter, we learned about dependency injection and services; in this chapter, we will learn how to write services, which will include methods so that we can make HTTP calls and process responses using `HttpClient`.

`HttpClient` is a small, easy-to-use, powerful library for performing HTTP request and response calls. Using `HttpClient`, we can easily communicate with backend services, and the module supports most modern web browsers. `HttpClient` comes with a lot of advanced functionality, such as interceptors and progress events. `HttpClient` supports various HTTP methods, including GET, POST, PUT, PATCH, DELETE, JSONP, and options. Each of these calls always returns an observable. We have to subscribe to the observables in order to process the responses. If we do not subscribe, nothing will happen.

`HttpClientModule` is available in the `@angular/common/http` library and needs to be imported into the `app.module.ts` file; otherwise, we will encounter errors.

We now know about the `HttpClient` module, but before we jump into implementing the module in our applications, it's good to know about some of the key functionality that got added to `HttpClient`:

- `HttpClient` provides a strongly typed response body.
- The request/response objects in `HttpClient` are immutable.
- The JSON format response is the default. We no longer have to map it into a JSON object.
- `HttpClient` provides interceptors that are really helpful in middleware for intercepting an `HttpRequest` for transforming or processing a response.
- `HttpClient` includes testability features. We can easily mock the requests and process headers more efficiently.

In the following section, we will learn about the `HttpClient` module, which needs to be imported into the component or service where we can make the HTTP calls. We will also learn about the HTTP verbs that are available and learn about their purpose in modern applications.

HttpClient and HTTP verbs

If the previous section was an introduction to `HttpClientModule` and `HttpClient` and its advantages, in this section, we will dig deeper and also learn how to write some sample code for implementing `HttpClient`.

As we mentioned earlier, `HttpClient` supports GET, POST, PUT, PATCH, DELETE, JSONP, and options methods, which will return observables. `HttpClient` also provides modules, which can easily pass various options and data using `HttpHeaders` and `HttpParams`.

In order to use `HttpClient`, we will need to import `HttpClientModule` into our application module (`app.module.ts`) file, and we also need to import `HttpClient` into our services or components and inject `HttpClient` inside the constructor so that we can use it to make HTTP calls. Add the following line of code to your `app.module.ts` file, and don't forget to add it to the list of import modules as well:

```
// Import the module into the component or service
import { HttpClient } from '@angular/core/http';

// Inside the constructor method inject the HttpClient and create an
instance
constructor(private http: HttpClient)
```

Now, let's implement some of the most frequently used HTTP verbs.



We will implement the HTTP methods separately for both the JSON server APIs and the Firestore database.

HTTP GET

We use the HTTP GET method to communicate with backend services to retrieve information from a particular URL resource. The sample code to get all listings is as follows:

```
getAllListings(): Observable<any>
{
  return this.http.get<Observable>('api/get-listing');
}
```

We have created a method named `getAllListings`, and we have explicitly mentioning the fact that the method will return an observable value of any data type. We have to pass the URL to the GET method. The URL is a mandatory value we need to pass. We can also pass optional data such as `Headers`, `Params`, `reportProgress`, and `responseType`. The GET method will return an instance of an RxJS observable, and we can subscribe to listen to the response.

On similar terms, we can easily create HTTP calls using the POST, PUT, and DELETE methods.

HTTP POST

Whenever we need to send any data or information securely to the server, such as a username, password, and email, we always use the POST method. The HTTP POST verb is always associated with creating or adding new data. It's secure and does not make data visible in the URL, unlike GET. In the POST method, along with the URL as a string, we will need to pass data to the URL. We can also pass options to the POST method, such as Headers, and Params. The following is the sample code for writing a sample HTTP POST call:

```
addNewListing(listing) {
    let httpHeaders = new HttpHeaders();
    httpHeaders.set('Content-Type', 'application/json');
    let options = { headers: httpHeaders};
    return this.http.post('api/add-listing', listing, options);
}
```

In the preceding code, we are creating a new method called `addNewListing`, which is accepting a param `listing`, which we will use as our data. We are creating an instance of `HttpHeaders`, so we create an object of the class, and we are setting the value of the `Content-Type` object to be `application/json`. We are then creating variable `options` and formatting them to send headers. Finally, we are making use of the `http.post` method to make a POST request.

HTTP PUT

In this section, we will learn how to make HTTP PUT calls. The PUT method is used to update or edit an existing dataset in the server. The HTTP PUT method involves a two-step process. First, we will need to retrieve data that we need to update and then pass the updated information back to the server using the POST method. The following is the sample code for creating the PUT method:

```
this.http.put(url, options);
```

We need to pass the URL as a mandatory parameter for the PUT method. Fortunately, there are various options available. For example, we can pass headers, params, and suchlike in the options.

HTTP DELETE

DELETE is an important operation of CRUD functionality. We can easily perform delete operations using the HTTP DELETE method. The `delete` operation can be achieved depending on the use case and the application's compliance. There are two types of deletions we can do—soft delete and hard delete:

- **Soft delete:** When using soft delete, we do not delete or erase the records from our database systems; instead, we update the records and set a column or field and mark it as deleted so that the records are not displayed to the user.
- **Hard delete:** The requested data is deleted permanently from the database system. Once the data is erased, it cannot be reverted or restored.

Let me give you a good example of both use cases. If you try to delete your Google account, it notifies you that you can come back and restore your account within x number of days, after which the data will be completely erased from their servers.

Back to our implementation. We can use the `http.delete()` method to implement the DELETE functionality in our applications. The sample code is given here:

```
this.http.delete(url, options);
```

We need to pass the URL value as mandatory and options, as the name implies, are optional for the `delete` method.

HTTP via promises

Promises are just a technical implementation of what real-world promises do! Suppose you had promised your boss that you would complete tasks assigned to you. If you do, that means a promise has been resolved and if you don't, it means it's been rejected. Similarly, a Promise in HTTP implementation means that we will wait for future data, either resolved or rejected, and then we'll do some logical processing based on the output received.

HTTP promises are a way to keep a placeholder for future data based on the success or failed states. Does this sound similar to regular HTTP calls? Yes, they are, with a major striking difference—*promises are asynchronous in nature*. When we make HTTP calls in Angular, it will wait until the request is completed and we receive the response; JavaScript will continue with the execution and, if it encounters synchronous assignments/operations, it will execute them immediately and fail if they are dependent on previous states or data.

A promise takes a callback method, which will take two parameters—`resolve` and `reject`. `resolve` means the method will return a promise object with a given message, while `reject` means the promise object is rejected with a reason. Then, you can expect `.then` and `.catch` to be called back if all goes well or not, respectively. The following is the sample code for writing a promise, showing the handling responses of `resolve` and `reject`:

```
//check if the listing status is active
ListingDetails(listing) {
  let promise = new Promise(function(resolve, reject) {
    if(listing.status == 'active') {
      resolved("listing is active");
    }
    else {
      reject("listing is not active");
    }

    promise.then((s => {
      //next steps after the promise has returned resolved
    }).catch((err => {
      // what to do when it's error or rejected
    })
  )
}
```

Let's analyze the preceding code in detail. We have implemented a promise and, as specified, the callback method will take two parameters, `resolve` and `reject`. We check whether the status of the listing is active; if yes, we resolve the promise; otherwise, we reject the promise. By default, the data returned by the `resolved` method will be passed to the `.then` method, and any failures or exceptions will be passed to the `.catch` method.

Since promises are asynchronous, which means we can chain events or methods, go ahead and add a method that will be called inside the `.then` method.

Awesome! We are now armed with all the theoretical knowledge about the classes and modules provided by Angular for HTTP functionality. We learned about `HttpClientModule`, `HttpClient`, and, above all, we learned about the various HTTP verbs we can make use of in our application. We also learned about HTTP observables and promises.

Now, it's time to get our hands dirty with code. We will learn how to create our multiple data sources, which we will need to integrate using HTTP calls. The first one will be using the fake JSON server APIs, while the second one will be using the Firestore database. In the next section, we will learn about, and create, the services that we will need before we start our mission of integrating functionality end to end.

Integrating backend services

We are making really good progress here, so let's keep rolling. One of the best practices in software development is to create code that is reusable, generic, and maintainable. In most applications that are dynamic in nature, we need to make a lot of HTTP calls to create, save, retrieve, edit, or delete data, as per the functional requirements of the application. If we do not have commonly shared HTTP calls, we may end up with a lot of methods having HTTP implementations, and it will be very difficult to maintain them in the long run. How do we address this situation? You already know the answer, my friend. That's right—by using services. In Chapter 11, *Dependency Injection and Services*, we learned all about Angular services and best practices regarding dependency injection.

Angular guidelines clearly state that all HTTP calls and functionality should be kept in services, which makes it easy to reuse existing code. Angular services are shared functions that allow us to access the properties and methods defined inside it. We will also create our custom services, in which we will implement our HTTP calls and which can be easily reused in various components. Let's create two services—one for working with JSON server APIs, and one for Firestore database operations. For working with JSON server APIs, we will call our `DbOperationsService` service, and for working with the Firestore database, we will call our `CRUDService` service. Each of these services will have the methods to make HTTP calls for creating, reading, updating, and deleting the data. Now, let's run the following `ng` command, which will generate our services:

```
ng generate service db-operations
```

Upon successful execution of the preceding command, we will execute the following command to generate another service. Let's call it `crud`. We will use the following `ng` command to generate the service.

```
ng generate service crud
```

Following a successful run, we should see the service files being generated, along with their respective spec files. So far, so good. We will need these services when we start the end-to-end integration work. It may look complicated, but trust me, all of it will make a lot of sense in the sections to follow.

Integrating Angular HTTP with backend APIs

This section is very important as this is the melting pot for most of the topics we have learned about throughout the course of this book. We are going to do complete end-to-end integration, from the UI to services, through to data sources.

We will need to generate the components that we are going to use in our application. Let's run the following `ng` commands to generate four components:

```
ng g component createListing  
ng g component viewListing  
ng g component deleteListing  
ng g component updateListing
```

When these commands are run successfully, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter12_backend\ListingApp>ng g c createListing  
CREATE src/app/create-listing/create-listing.component.html (33 bytes)  
CREATE src/app/create-listing/create-listing.component.spec.ts (678 bytes)  
CREATE src/app/create-listing/create-listing.component.ts (301 bytes)  
CREATE src/app/create-listing/create-listing.component.scss (0 bytes)  
UPDATE src/app/app.module.ts (505 bytes)  
  
D:\book_2\book\chapter12_backend\ListingApp>ng g c viewListing  
CREATE src/app/view-listing/view-listing.component.html (33 bytes)  
CREATE src/app/view-listing/view-listing.component.spec.ts (664 bytes)  
CREATE src/app/view-listing/view-listing.component.ts (293 bytes)  
CREATE src/app/view-listing/view-listing.component.scss (0 bytes)  
UPDATE src/app/app.module.ts (609 bytes)  
  
D:\book_2\book\chapter12_backend\ListingApp>ng g c deleteListing  
CREATE src/app/delete-listing/delete-listing.component.html (33 bytes)  
CREATE src/app/delete-listing/delete-listing.component.spec.ts (678 bytes)  
CREATE src/app/delete-listing/delete-listing.component.ts (301 bytes)  
CREATE src/app/delete-listing/delete-listing.component.scss (0 bytes)  
UPDATE src/app/app.module.ts (721 bytes)  
  
D:\book_2\book\chapter12_backend\ListingApp>ng g c updateListing  
CREATE src/app/update-listing/update-listing.component.html (33 bytes)  
CREATE src/app/update-listing/update-listing.component.spec.ts (678 bytes)  
CREATE src/app/update-listing/update-listing.component.ts (301 bytes)  
CREATE src/app/update-listing/update-listing.component.scss (0 bytes)  
UPDATE src/app/app.module.ts (833 bytes)
```

Now that we have generated our components, we will make use of the DbOperationsService service we generated in the previous section. We will also use our fake APIs that we created using the JSON server. We will implement methods for getting all the listings, viewing a particular listing, editing an existing listing, and finally, deleting a listing. In order to achieve this, we will need to import HttpClientModule into our app.module.ts file. We will also need to import HttpClient into our db-operations.service.ts service file. We will also import the HttpHeaders module. This is not mandatory, but, by way of good practice, we will be importing and using it while making our HTTP calls. We will be adding the following code to the db-operations.service.ts file:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders, HttpParams } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class DbOperationsService {

  constructor(private http: HttpClient) { }

  getListings(){
    return this.http.get('http://localhost:3000/listings');
  }
  viewListing(id){
    return this.http.get('http://localhost:3000/listings/'+id);
  }
  addListing(newList){
    let headers = new HttpHeaders({ 'Content-Type': 'application/json' });
    return this.http.post('http://localhost:3000/listings', newList);
  }
  editListing(id, newList){
    let headers = new HttpHeaders({ 'Content-Type': 'application/json' });
    return this.http.put('http://localhost:3000/listings/'+id, newList);
  }
  deleteListing(id){
    return this.http.delete('http://localhost:3000/listings/'+id);
  }
}
```

Let's analyze the preceding code in detail. First, we are importing the required modules: `Injectable`, `HttpClient`, `HttpHeaders`, and `HttpParams`. We are then injecting `HttpClient` into our constructor and creating an instance named `http`. We are then creating four methods, namely; `getListings`, `viewListing`, `editListing`, and `deleteListing`. In the `getListings` method, we are calling the API URL using the HTTP GET method. This will return all the listings from the `data.json` file we created earlier. In `viewListing`, we pass the ID of the Listing to retrieve the data of the listing using the HTTP GET method. In the `addListing` method, we are calling the API and passing the data object using the HTTP POST method. This will create a new row in our JSON file. Next up is the `editListing` method, which takes two parameters—the ID of the listing and the updated data object, which we need to save. The last method is `deleteListing`, to which we will pass the ID of the listing we want to delete.



In a more practical world, we would need to pass authentication tokens, additional security, cleaning data, and so on.

We have now made our custom service, which includes the methods that will make HTTP calls. Before we start working on our components, we will create a few routes where we will map the components we have generated. Open the `app-routing.module.ts` file and import all our components inside it. Then, we will need to add the routes to it, as shown in the following code block:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {UpdateListingComponent} from './update-listing/update-listing.component';
import {CreateListingComponent} from './create-listing/create-listing.component';
import {ViewListingComponent} from './view-listing/view-listing.component';
import {DeleteListingComponent} from './delete-listing/delete-listing.component';

const routes: Routes = [
  {path:'create-listing', component:CreateListingComponent },
  { path:'view-listing', component:ViewListingComponent },
  { path:'delete-listing/:id', component:DeleteListingComponent},
  {path:'update-listing/:id', component:UpdateListingComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule { }
```

In the preceding code, we are updating our AppRoutingModule and adding five routes. We created the create-listing and view-listing routes and mapped them to CreateListingComponent and ViewListingComponent, respectively. That's very straightforward. For the delete-listing and update-listing routes, notice that we are passing a parameter named ID. We will use these and pass the listing ID in order to delete or update the data for a listing.

Now that we have created our service and routes, they are ready to be implemented in our components. Let's start working on our components. First, we will start with ViewListComponent. Open the view-listing.component.ts file and add functionality to retrieve all the listings, as shown in the following code block:

```
import { Component, OnInit } from '@angular/core';
import { DbOperationsService } from '../db-operations.service';
import { Listing } from '../models/listing';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-view-listing',
  templateUrl: './view-listing.component.html',
  styleUrls: ['./view-listing.component.scss']
})

export class ViewListComponent implements OnInit {

  listArr: Observable<any[]>;
  viewList: Observable<Listing>;
  isViewPage: boolean = false;

  constructor(private dbOps: DbOperationsService) { }

  ngOnInit() {
    this.dbOps.getListings().subscribe((data) => {this.listArr = data});
  }

  showListing(listing) {
    this.isViewPage = true;
    this.dbOps.viewListing(listing.id).subscribe((data) => {this.viewList = data});
  }
}
```

Let's analyze the preceding code in detail. First, we need to import all the required modules and classes. We are importing `DbOperationsService` we created. We are also importing the listing interface class we created previously. Since we will be working with the `Listing` interface class, we will need to import `Observable` from `rxjs`. Next, we are declaring our selector as `app-view-listing`; we will call this directive in the template `view-listing.component.html` file. We will now create three variables, named `listArr`, `viewList`, and `isViewPage`. Note that `listArr` and `viewList` are declared as `Observable`. The difference between the `listArr` and `viewList` variables is that `listArr` is an observable of the `Listing` type and is an array, whereas `viewList` is an `Observable` of the `Listing` type and will hold a single list value. Since we have imported a service, we will need to create an instance called `dbOps` in our constructor method. We will implement the `ngOnInit` method here; we are calling the `getListings` method using the instance of the `dbOps` service. We are subscribing to the method, which means we will map the data to the `listArr` variable. We will then use the `listArr` variable to display it in the template file. Finally, we are creating a `showListing` method to which we are passing the listing object. Using the instance of the service, we are calling the `viewListing` method and passing the listing ID. We are subscribing to the data and mapping it to the `viewList` variable.

Now, we need to update our template in the `view-listing.component.html` file and use the `listArr` and `viewList` variables to display the data in the page, as shown in the following code block:

```
<h4>Show All Listings</h4>






```

In the preceding code, we have created a table. Using `ngFor`, we are looping the data that we get from the API, and, using interpolation, we are displaying the data in the table rows. Note that, for the anchor tags, we are using the `routerLink` directive to dynamically create the link, and we are passing the ID for the edit and delete links.

I am sure you are excited about the end result. Let's run the `ng serve` command. You should see the following output:

| Title | Description | Price | Status | Actions |
|-------------------------|---------------------------------------|-------|----------|---|
| Dawn at Miami | Walking down the road in Miami | 150 | Active | Edit Delete |
| Sunrise in New York | Welcome to New York City123 | 167 | Active | Edit Delete |
| Sunset in California | Summer of 69 and sunset in California | 134 | Active | Edit Delete |
| Breakfast in California | Eggs and Cheese is my favorite | 20 | Active | Edit Delete |
| An Evening in Honk Kong | Hong Kong is a beautiful place | 320 | Inactive | Edit Delete |

Beautiful! Now things are really cooking! There's no better encouragement than seeing the code in action. We have added the **Add New Listing** menu link, so now it's time to implement that functionality in our `createListing` component.

Open `createListingComponent` and modify the `create-listing.component.ts` file by adding the following code to it:

```
import { Component, OnInit } from '@angular/core';
import { DbOperationsService } from '../db-operations.service';

@Component({
  selector: 'app-create-listing',
  templateUrl: './create-listing.component.html',
  styleUrls: ['./create-listing.component.scss']
})

export class CreateListingComponent implements OnInit {
  userId = 1;
  newListing;
  successMsg;
```

```
constructor(private dbOps: DbOperationsService) { }

ngOnInit() {
}
addNewList(listForm)
{
  this.newListing = {
    "userId":this.userId,
    "id": 152,
    "title":listForm.title,
    "price":listForm.price,
    "status":listForm.status,
  };

  this.dbOps.addListing(this.newListing).subscribe((data) => {
    this.successMsg = data;
  });
}
}
```

Let's analyze the preceding code in detail. We are importing the required modules in the file. We are also importing the `DbOperationsService`, which we created earlier. We are creating a few variables; that is, `userId`, `newListing`, and `successMsg`, and assigning some initial values. We are creating an `addNewList` method, and we are passing the `listForm` data. We are also creating a data structure similar to our listing model that we created. Next, using the instance of the service, we are calling the `addListing` method and passing the data object that we need to save. This will create a new record in our `data.json` file. Finally, we are mapping the result to the `successMsg` variable. We will use this to display the success message to the user.



Since we are using a fake API, we have stubbed the value of the ID. In a more real-time scenario, this ID will be auto-incremented on the database side and would always be a unique value.

Now, it's time to update our template file so that we can get data from the user using the form. Open the `create-listing.component.html` file and add the following code to it:

```
<h4>Add New Listing</h4>
<p>
<div class="container">

<div *ngIf="successMsg">List Added Successful</div>

<form #listingForm="ngForm" (ngSubmit)="addNewList(listingForm)">
  <div class="form-group">
    <label for="title">Enter Listing Title</label>
    <input type="text" [ngModel]="title" name="title" class="form-control"
      placeholder="Enter title">
  </div>
  <div class="form-group">
    <label for="price">Enter Description</label>
    <input type="text" [ngModel]="description" name="description"
      class="form-control" placeholder="Enter Description">
  </div>
  <div class="form-group">
    <label for="price">Enter Price</label>
    <input type="number" [ngModel]="price" name="price" class="form-control"
      placeholder="Enter price here">
  </div>
  <div class="form-group form-check">
    <input type="checkbox" [ngModel]="status" name="status"
      class="form-check-input">
    <label class="form-check-label" for="status">Active?</label>
  </div>
  <button type="submit" class="btn btn-primary">Add New Listing</button>
</form>
</div>
```

In the preceding code, we are creating a form using the template-driven forms. We have created a few form fields to capture data, such as title, description, price, and active. We are using the template variables for the form and the fields. We are also calling the `addNewList` method on the `ngSubmit` event and submitting the entire form. By running the `ng serve` command, we should see the following output:

The screenshot shows a web browser window with the URL `localhost:4200/create-listing`. The page title is "ListingApp". The main content area has a yellow header bar with the text "Add New Listing". Below it, there are three input fields: "Enter Listing Title" with placeholder "Enter title", "Enter Description" with placeholder "Enter Description", and "Enter Price" with placeholder "Enter price here". There is also a checkbox labeled "Active?" and a blue "Add New Listing" button.

Now, go ahead and add some data to the form fields and then click on the **Submit** button. You should see a success message if the record has been created successfully:

The screenshot shows the same web browser window after a submission. A green success message box at the top says "List Added Successful" with a close button "x". The form fields now contain the data entered: "Welcome to Disney World" in the title field, "Disney is not only for kids!" in the description field, and "600" in the price field. The "Active?" checkbox is checked. The "Add New Listing" button remains visible at the bottom.

Now, click on the **Get All Listings** link in the menu. You should see the newly created record in the listings displayed in the table. Do you remember that we added the edit and delete links for the listings? It's time to implement them now. We will start with the edit functionality first and then implement the delete functionality.

Open our update listing component, edit the `update-listing.component.ts` file, and then add the following code to it:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from "@angular/router";
import { DbOperationsService } from '../db-operations.service';
import { Listing } from '../models/listing';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-update-listing',
  templateUrl: './update-listing.component.html',
  styleUrls: ['./update-listing.component.scss']
})
export class UpdateListingComponent implements OnInit {

  listId;
  successMsg = false;
  viewList: Observable<Listing>;

  constructor(private route:ActivatedRoute, private
    dbOps:DbOperationsService) { }

  ngOnInit() {
    this.listId = this.route.snapshot.paramMap.get("id");
    this.dbOps.viewListing(this.listId).subscribe((data)
      => {this.viewList = data});
  }
  editListing(updatedList){
    this.dbOps.editListing(updatedList.id, updatedList).subscribe((data) =>
{
    this.successMsg = data;
  });
}
}
```

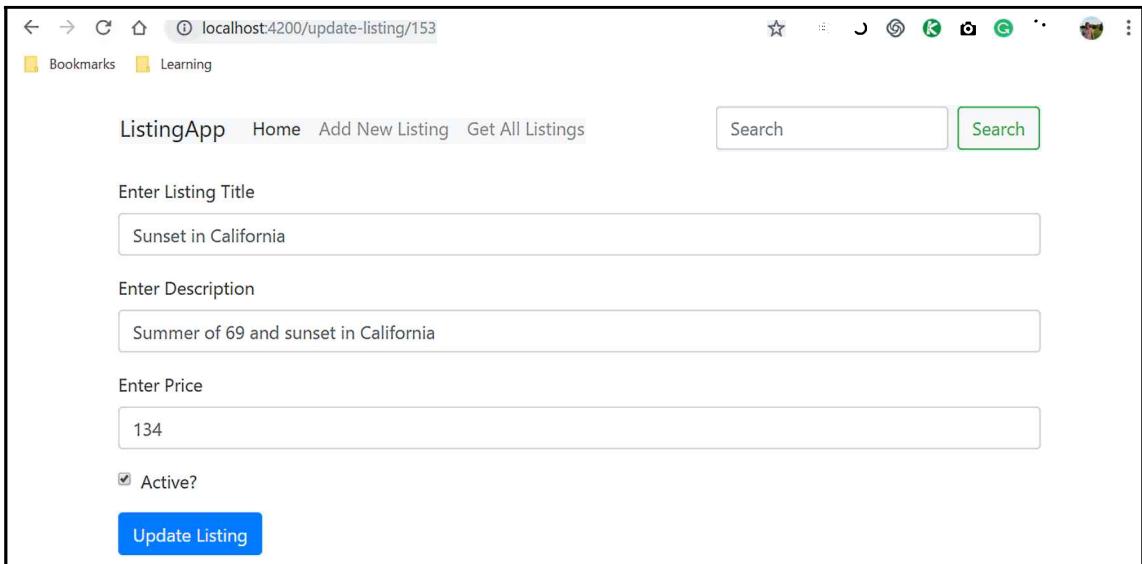
Let's analyze the preceding code in detail. We are importing the required modules into our component file. We are importing `ActivatedRoute`, our service, listing interface class, and observable into the component file. In order to achieve update functionality, we need to do two things. First, we will need to retrieve the data of the listing for which the ID is passed. Once the user updates the data and clicks on the `Submit` button, we will persist the data for that listing. We will also need to inject the router and service into our constructor. On the `ngOnInit` method, using the router snapshot, we are capturing the ID of the listing from the URL. Then, using the instance of the service, we are calling the `viewListing` method to get details of the listing based on the ID that's passed. Finally, we have created an `editListing` method. Using the instance of the service, we are calling the `editListing` method, and so we need to pass two parameters, one for passing the ID of the listing, and another for passing the updated data of the listings.

Now, let's update our template file. Open the `update-listing.component.html` file and add the following code:

```
<div class="container">
<div *ngIf="successMsg">List Updated Successful</div>
<form #editlistingForm="ngForm" (ngSubmit)="editListing(editlistingForm)">
  <div class="form-group">
    <input type="hidden" class="form-control" name="id"
      [(ngModel)]="viewList.id" ngModel #id>
  </div>
  <div class="form-group">
    <input type="hidden" class="form-control" name="userId"
      [(ngModel)]="viewList.userId" ngModel #userId>
  </div>
  <div class="form-group">
    <label for="title">Enter Listing Title</label>
    <input type="text" class="form-control" name="title"
      [(ngModel)]="viewList.title" ngModel #title required>
  </div>
  <div class="form-group">
    <label for="price">Enter Description</label>
    <input type="text" name="description" [(ngModel)]="viewList.description"
      ngModel #description class="form-control" required>
  </div>
  <div class="form-group">
    <label for="price">Enter Price</label>
    <input type="number" [(ngModel)]="viewList.price" name="price"
      class="form-control" ngModel #price required>
  </div>
  <div class="form-group form-check">
    <input type="checkbox" [(ngModel)]="viewList.status"
      checked="{{viewList.status}}" name="status" ngModel
      #status class="form-check-input" required>
```

```
<label class="form-check-label" for="status">Active?</label>
</div>
<button type="submit" [disabled]="!editListingForm.valid"
        class="btn btn-primary">Update Listing</button>
</form>
</div>
```

In the preceding code, we are once again creating a form based on the template-driven form approach. You will notice that the edit form is very much similar to the create listing form. You are almost correct, but there are some important differences. Note that we are now using two-way data binding with `ngModel` and binding the value to the form field. With this, when we get the initial data, it's displayed in the form field. Now, the user can edit data and, when clicking on the **Update Listing** button, the data is sent to the `addListing` method and persisted in the backend API. Now, let's see it in action. By running the `ng serve` command, we should see the following output:



Notice that the URL has the ID of the listing that was passed as the parameter. The data is retrieved and has been displayed on the page load. Now, when the user updates the details in the form and clicks on the **Submit** button, this will update the data of the listings. That's your homework.

Alright, so we have implemented create, edit, and view functionality. Next, we will implement the delete functionality of the listings. Remember that, for the delete and edit functionalities, the user will always navigate to the pages on click-through anchor tags. Open DeleteListingComponent and update the delete-listing.component.ts file, as shown in the following code block:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from "@angular/router";
import { DbOperationsService} from '../db-operations.service';
import { Listing} from '../models/listing';
import {Observable} from 'rxjs';

@Component({
  selector: 'app-delete-listing',
  templateUrl: './delete-listing.component.html',
  styleUrls: ['./delete-listing.component.scss']
})
export class DeleteListingComponent implements OnInit {
  viewList:Observable<Listing>;
  listId;
  successMsg:Observable<Listing>;

  constructor(private route:ActivatedRoute, private dbOps:DbOperationsService) { }

  ngOnInit() {
    this.listId = this.route.snapshot.paramMap.get("id");
    this.dbOps.deleteListing(this.listId).subscribe((data) => {
      this.successMsg = data;
    });
  }
}
```

Let's analyze the preceding code in detail. We are importing the required modules in the component file; that is, ActivatedRoute, DbOperationsService, Listing, and Observable. We are also creating a few variables—viewList, ListId, and successMsg. Then, we are injecting the route and service into the constructor method. Finally, with the ngOnInit method, we are passing the ID of the listing that needs to be deleted. We are subscribing the data and mapping it to successMsg.

In this section, we have learned how to implement basic CRUD operations for our ListingApp. Then, we learned how to make HTTP calls for the GET, POST, PUT, and DELETE methods. Finally, we learned how to create fake APIs using the JSON Server. In the next section, we will learn how to implement CRUD operations using the cloud NoSQL Firestore database.

Integrating Angular HTTP with Google Firebase

In this section, we will learn how to implement HTTP functionality for a NoSQL Firestore database. We created our Firestore database in an earlier section. Now is the right time to integrate the Angular HTTP calls, which will invoke and work with the Firestore database.

What are the use cases we will implement? For our ListingApp, we will need a commenting system. As a user, we should be able to add, edit, delete, and view comments. All of these use cases will require us to make HTTP calls to APIs to save, retrieve, and delete comments.

Angular Fire is the official library for Firebase. The library provides a lot of built-in modules that support activities such as authentication, working with Firestore databases, observable-based push notifications, and much more.

We will need to install this module under `@angular/fire`. Run the following command in the command-line interface to install the library:

```
npm i @angular/fire
```

When we run the preceding command successfully, we should see the following output:

```
D:\book_2\book\chapter12_backend\ListingApp>npm i @angular/fire
npm WARN @angular/fire@5.1.1 requires a peer of firebase@5.5.0 but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.7 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ @angular/fire@5.1.1
updated 1 package and audited 41387 packages in 17.828s
found 0 vulnerabilities
```

Once we have installed the library, we will proceed and create a new custom service for our integration pieces with the Firestore database.

Run the following command to generate a new service:

```
ng generate service crudService
```

When we run the preceding command successfully, we should see the following output:

```
D:\book_2\book\chapter12_backend\ListingApp>ng generate service crudService
CREATE src/app/crud-service.service.spec.ts (359 bytes)
CREATE src/app/crud-service.service.ts (140 bytes)
```

You will notice that two files are generated. We will implement all our HTTP calls inside the service. As we mentioned previously, we will need to create a few components that will map to each piece of functionality and will internally call the service that has the HTTP implementations.

Run the following `ng generate` commands to generate components for the comment's functionality:

```
ng generate component addComments

ng generate component viewComments

ng generate component editComments

ng generate component deleteComments
```

When we run the preceding commands successfully, we should see the following output:

```
D:\book_2\book\chapter12_backend\ListingApp>ng generate component addComments
CREATE src/app/add-comments/add-comments.component.html (33 bytes)
CREATE src/app/add-comments/add-comments.component.spec.ts (664 bytes)
CREATE src/app/add-comments/add-comments.component.ts (293 bytes)
CREATE src/app/add-comments/add-comments.component.scss (0 bytes)
UPDATE src/app/app.module.ts (1467 bytes)

D:\book_2\book\chapter12_backend\ListingApp>ng generate component editComments
CREATE src/app/edit-comments/edit-comments.component.html (33 bytes)
CREATE src/app/edit-comments/edit-comments.component.spec.ts (671 bytes)
CREATE src/app/edit-comments/edit-comments.component.ts (297 bytes)
CREATE src/app/edit-comments/edit-comments.component.scss (0 bytes)
UPDATE src/app/app.module.ts (1575 bytes)

D:\book_2\book\chapter12_backend\ListingApp>ng generate component viewComments
CREATE src/app/view-comments/view-comments.component.html (33 bytes)
CREATE src/app/view-comments/view-comments.component.spec.ts (671 bytes)
CREATE src/app/view-comments/view-comments.component.ts (297 bytes)
CREATE src/app/view-comments/view-comments.component.scss (0 bytes)
UPDATE src/app/app.module.ts (1683 bytes)

D:\book_2\book\chapter12_backend\ListingApp>ng generate component deleteComments
CREATE src/app/delete-comments/delete-comments.component.html (33 bytes)
CREATE src/app/delete-comments/delete-comments.component.spec.ts (685 bytes)
CREATE src/app/delete-comments/delete-comments.component.ts (305 bytes)
CREATE src/app/delete-comments/delete-comments.component.scss (0 bytes)
UPDATE src/app/app.module.ts (1799 bytes)
```

You will notice that the components have been generated and added to our project directory. You will also notice that the `app.module.ts` file has been updated with entries for the components.

We have generated our components and the service that's required for our integration. We have also installed the Angular Fire library. In order to use the Angular Fire library in our application, we will need to import the library into our `app.module.ts` file. Import the required modules into the app module file and list the modules in the import list of our application, as shown here:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CreateListingComponent } from './create-listing/create-listing.component';
import { ViewListingComponent } from './view-listing/view-listing.component';
import { DeleteListingComponent } from './delete-listing/delete-listing.component';
import { UpdateListingComponent } from './update-listing/update-listing.component';

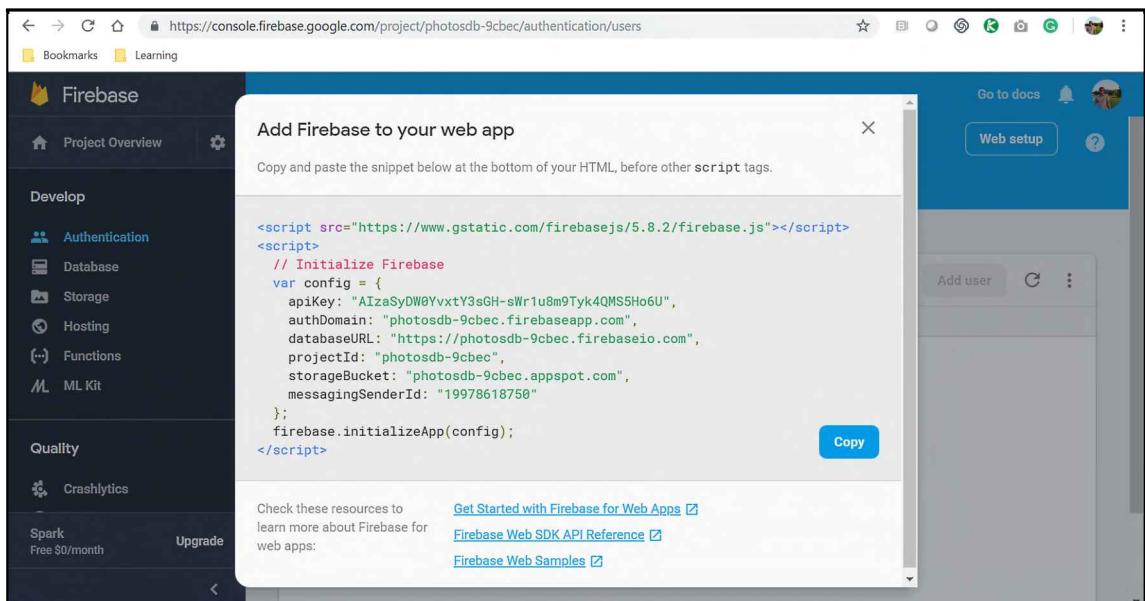
import {FormsModule} from '@angular/forms';

import { AngularFireModule} from 'angularfire2';
import {AngularFireDatabaseModule} from 'angularfire2/database';
import { AngularFireAuth } from '@angular/fire/auth';
import { environment } from './firebase-config';
import { AngularFirestore } from '@angular/fire/firestore';
import { AddCommentsComponent } from './add-comments/add-comments.component';
import { EditCommentsComponent } from './edit-comments/edit-comments.component';
import { ViewCommentsComponent } from './view-comments/view-comments.component';
import { DeleteCommentsComponent } from './delete-comments/delete-comments.component';

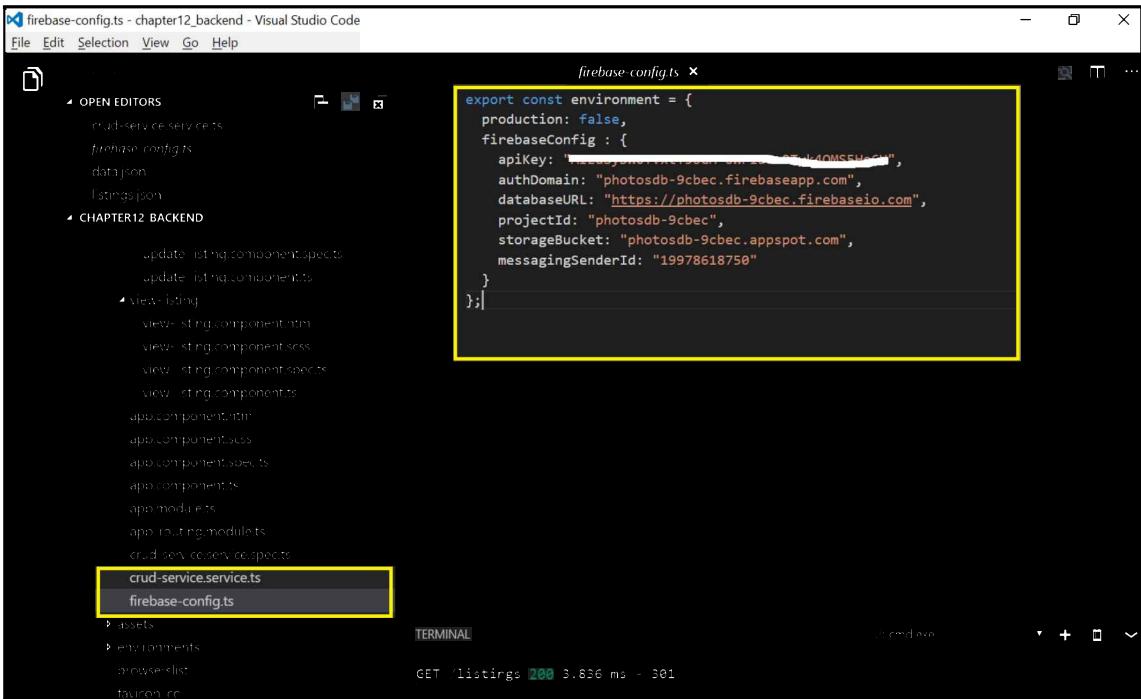
@NgModule({
  declarations: [
    AppComponent,
    CreateListingComponent,
    ViewListingComponent,
    DeleteListingComponent,
```

```
UpdateListComponent,
AddCommentsComponent,
EditCommentsComponent,
ViewCommentsComponent,
DeleteCommentsComponent
],
imports: [
  BrowserModule,
  HttpClientModule,
  AppRoutingModule,
  AngularFireModule.initializeApp(environment.firebaseioConfig),
  AngularFireDatabaseModule,
  FormsModule
],
providers: [AngularFirestore],
bootstrap: [AppComponent]
})
export class AppModule { }
```

One important thing to note in the preceding code is that we are importing the required modules from Angular Fire and also listing them under the import module list. Notice that we have imported a file called `firebase-config`. These are environment variables, which will hold the API keys for authentication with Firebase. We can find the API keys listed under the Firebase account, as shown in the following screenshot:



We will need to copy the details into the `firebase-config.ts` file. The following screenshot displays the settings specified in our ListingApp:



```
export const environment = {
  production: false,
  firebaseConfig : {
    apiKey: "AIzaSyCvWzXGJLjDfIwOOGmZM7HgkVQcJyPQ",
    authDomain: "photosdb-9cbe9.firebaseio.com",
    databaseURL: "https://photosdb-9cbe9.firebaseio.com",
    projectId: "photosdb-9cbe9",
    storageBucket: "photosdb-9cbe9.appspot.com",
    messagingSenderId: "19978618750"
  }
};
```

So far, so good. Now that we have installed the required library, imported the modules, and done the configuration settings, it's time to work on our application components. We are making great progress here. Let's keep this momentum going.

Now that we have created our components, we will quickly modify our `app-routing.module.ts` file and create a new route for each of these components.



We have already mastered Angular routing, in Chapter 4, *Routing*. Revisit that chapter if you need a quick refresher.

In the following code, we have imported all the required component classes into the `app-routing.module.ts` file and added the respective routes to the routing file:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {UpdateListingComponent} from './update-listing/update-
```

```
listing.component';
import {CreateListingComponent} from './create-listing/create-
listing.component';
import {ViewListingComponent} from './view-listing/view-listing.component';
import {DeleteListingComponent} from './delete-listing/delete-
listing.component';

import { AddCommentsComponent } from './add-comments/add-
comments.component';
import { EditCommentsComponent } from './edit-comments/edit-
comments.component';
import { ViewCommentsComponent } from './view-comments/view-
comments.component';
import { DeleteCommentsComponent } from './delete-comments/delete-
comments.component';

const routes: Routes = [
  { path:'create-listing', component:CreateListingComponent },
  { path:'view-listing', component:ViewListingComponent },
  { path:'delete-listing/:id', component:DeleteListingComponent},
  { path:'update-listing/:id', component:UpdateListingComponent},
  { path:'add-comment', component:AddCommentsComponent },
  { path:'view-comment', component:ViewCommentsComponent },
  { path:'delete-comment/:id', component:DeleteCommentsComponent},
  { path:'update-comment/:id', component>EditCommentsComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

We will use the four newly created routes to implement the comment's functionality in ListingApp. We are going to add the CRUD operations using the Firestore database. We will need to import the AngularFirestore module to our service, as shown here:

```
import { AngularFirestore } from '@angular/fire/firestore';
```

After we have imported the module into our file, we will need to inject it inside the constructor method, as follows:

```
constructor(private afStore : AngularFirestore, private route: Router ) { }
```

We can now make use of the `AngularFirestore` module and implement CRUD operations using Firestore. Take a look at the complete updated code in the `crud-service.service.ts` file:

```
import { Injectable } from '@angular/core';
import { AngularFireAuth } from '@angular/fire/auth';
import { environment } from './firebase-config';
import { AngularFirestore } from '@angular/fire/firestore';

@Injectable({
  providedIn: 'root'
})
export class CrudServiceService {

  constructor(private afStore : AngularFirestore) { }

  getComments() {
    return this.afStore.collection('comments');
  }

  deleteComment(id) {
    this.afStore.collection('comments').doc(id).delete();
  }

  addComment(newComment) {
    this.afStore.collection('comments').add(newComment);
  }

  updateComment(id, editedComment) {
    this.afStore.collection('comments').doc(id).set(editedComment);
  }
}
```

Let's analyze the preceding code in detail. We have imported all the required modules, including our Angular Fire module and our `firebase-config` file. Since we have imported our `AngularFireStore` module, we will need to inject it into our `constructor` method and create an instance of it. We are creating methods for each of the actions for the comment's functionality. In the `getComments` method, we are retrieving all the data from the `comments` collection. In the `deleteComment` method, we are passing the ID of the comment we need to delete. In the `addComment` method, we are passing the data that we want to store in our collection. In the `updateComment` method, we are passing two parameters; the first is the ID of the comment we want to update, and the second is the updated data that we need to persist in the database.

You may wonder why we did not make any HTTP calls in these methods? The `AngularFireStore` module internally makes HTTP calls to the service and will authenticate and get account-specific information from the firebase config file.

In earlier sections, we learned how to send data from components to the service, right? Along the same lines, go ahead and try for comments functionality. That's your homework.

Summary

How do you feel? You should feel great and you should be proud of yourself! This chapter was a lot of work, but we're better off for having done it. It brought together all the aspects we have learned so far, such as forms, components, routing, services, and more.

For frontend developers, having a fake API setup in the local development environment always helps us to work independently without depending on backend developers or APIs. We learned about building fake APIs using the JSON server. We learned about the NoSQL document database, particularly the Firestore database provided by Google Cloud. We deep dived into Angular HTTP concepts and functionalities. We learned how to make HTTP POST, GET, PUT and DELETE calls. We also implemented our entire application's functional use cases using both the JSON Server and Firestore databases.

We have made tremendous progress so far. We are now capable of developing Angular applications end to end, utilizing all the superpowers that Angular provides, including forms, components, services, routing, and much more. At the end of this chapter, I am confident that we are able to bring together all the pieces of the Angular framework into a single working app.

Having a working application up and running is a good sign of progress. But the important factor in terms of judging the application is to look at the quality checks or unit tests.

In the next chapter, we are going to learn how to write unit tests to make sure we catch any defects early in the product development life cycle. Writing test scripts ensures quality and is a great sign of handling all use cases, including both the happy and negative paths of our application.

13

Unit Testing

You've probably written unit tests for traditional server-side code, such as for Java, Python, or C#. Unit testing is, of course, just as important on the client side, and, in this chapter, you will learn about Angular testing, including the Jasmine and Karma frameworks, two excellent tools for unit testing your client-side code.

Together, we'll explore how we can unit test various parts of our Angular application, such as our components, routes, and **dependency injection (DI)**.

This chapter will cover the following topics:

- An introduction to Jasmine and Karma
- Testing directives
- Testing components
- Testing routing
- Testing dependency injection
- Testing HTTP

Introduction to testing frameworks

In this section, we will learn about two important testing frameworks, namely Jasmine and Karma.

Testing is as important as development itself. It's a highly debatable topic, with some experts believing in **test-driven development (TDD)**, which means that writing test scripts is important even before we write development code.

The beauty about Angular framework is that it natively supports testing frameworks and offers a lot of testing utilities that make the developer's job happy and easy. We are not complaining at all.

Angular provides us with a core testing module, which has a lot of awesome classes we can make use of, and natively supports two important testing frameworks, namely Jasmine and Karma:

- We write our test scripts using the Jasmine framework.
- We use the Karma framework to execute the test scripts.

About the Jasmine framework

Jasmine is a leading open source testing framework for writing and testing automated test scripts for modern web frameworks.

Certainly, for Angular, Jasmine has become the de facto, go-to framework. The following is taken from the official website:

"Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests."

The idea behind writing Jasmine test scripts are behaviorally and functionally driven. Test scripts have two important elements—describe and the specs (it):

- The describe function is for grouping related specs together.
- The specs are defined by calling the it function.

Here's a sample test script, which is written in Jasmine:

```
describe("Test suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

In the process of writing test specs, we have to use a lot of conditional checks to match data, elements, results, asserting conditions, and much more. The Jasmine framework provides a lot of matchers, which we can readily use while writing our test specs. In the preceding sample code, toBe is one such example of a matcher.

Here's a list of the most commonly and frequently used matchers in Jasmine:

- toBe
- toBeTruthy
- toBeFalsy
- toBeGreaterThanOrEqual
- toBeLessThanOrEqual
- toHaveBeenCalled
- toHaveClass
- toMatch

We will learn how to use these matchers in the next few sections. OK, we have written our test specs, so now what? How do we run them? What will run them for us? The answers can be found in the next section.

About the Karma framework

Karma is a test-runner framework for executing test scripts on a server and generating the reports.

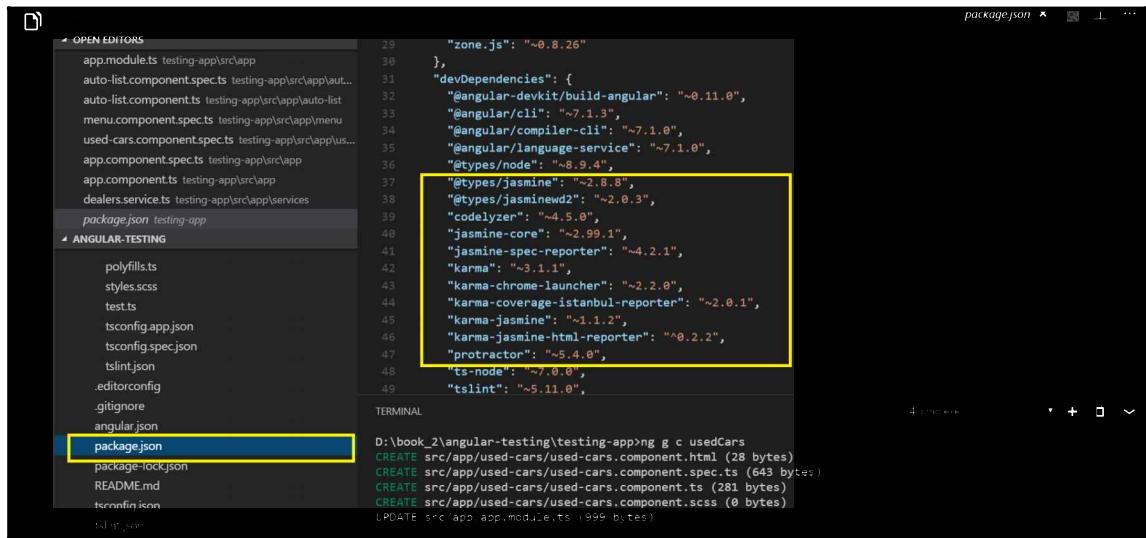
The following is taken from the official website:

"Karma is essentially a tool which spawns a web server that executes source code against test code for each of the browsers connected. The results of each test against each browser are examined and displayed via the command line to the developer such that they can see which browsers and tests passed or failed."

The Karma framework gets added in our list of dependencies as it is included in the Angular CLI installation. Before we proceed to write and execute our test scripts, it's good practice to verify whether we have installed both Jasmine and Karma correctly in our package.json file. We can also verify the version numbers of the libraries that are being used.

I bet you figured out that this is also the place to specify any particular version of Jasmine and Karma you want to use.

In the following screenshot, we can verify that we have added Jasmine and Karma to our list of devDependencies in our package.json file:



The screenshot shows a code editor with the package.json file open. A yellow box highlights the devDependencies section of the JSON object. The terminal below shows the command 'ng g c usedCars' being run, which generates several files in the src/app/used-cars directory.

```
29   "zone.js": "~0.8.26"
30 },
31 "devDependencies": {
32   "@angular-devkit/build-angular": "~0.11.0",
33   "@angular/cli": "~7.1.3",
34   "@angular/compiler-cli": "~7.1.0",
35   "@angular/language-service": "~7.1.0",
36   "@types/node": "~8.9.4",
37   "@types/jasmine": "~2.8.8",
38   "@types/jasminewd2": "~2.0.3",
39   "codelyzer": "~4.5.0",
40   "jasmine-core": "~2.99.1",
41   "jasmine-spec-reporter": "~4.2.1",
42   "karma": "~3.1.1",
43   "karma-chrome-launcher": "~2.2.0",
44   "karma-coverage-istanbul-reporter": "~2.0.1",
45   "karma-jasmine": "~1.1.2",
46   "karma-jasmine-html-reporter": "~0.2.2",
47   "protractor": "~5.4.0",
48   "ts-node": "~7.0.0",
49   "tslint": "~5.11.0",
50 }
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
678
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
768
769
769
770
771
772
773
774
775
776
777
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
847
847
848
849
849
850
851
852
853
854
855
856
857
857
858
859
859
860
861
862
863
864
865
866
867
867
868
869
869
870
871
872
873
874
875
876
876
877
878
878
879
879
880
881
882
883
884
885
886
886
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
905
906
907
907
908
909
909
910
911
912
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640
1641
1641
1642
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1650
1651
1651
1652
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1660
1661
1661
1662
1662
1663
1663
1664
1664
1665
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1670
1671
1671
1672
1672
1673
1673
1674
1674
1675
1675
1676
1676
1677
1677
1678
1678
1679
1679
1680
1680
1681
1681
1682
1682
1683
1683
1684
1684
1685
1685
1686
1686
1687
1687
1688
1688
1689
1689
1690
1690
1691
1691
1692
1692
1693
1693
1694
1694
1695
1695
1696
1696
1697
1697
1698
1698
1699
1699
1700
1700
1701
1701
1702
1702
1703
1703
1704
1704
1705
1705
1706
1706
1707
1707
1708
1708
1709
1709
1710
1710
1711
1711
1712
1712
1713
1713
1714
1714
1715
1715
1716
1716
1717
1717
1718
1718
1719
1719
1720
1720
1721
1721
1722
1722
1723
1723
1724
1724
1725
1725
1726
1726
1727
1727
1728
1728
1729
1729
1730
1730
1731
1731
1732
1732
1733
1733
1734
1734
1735

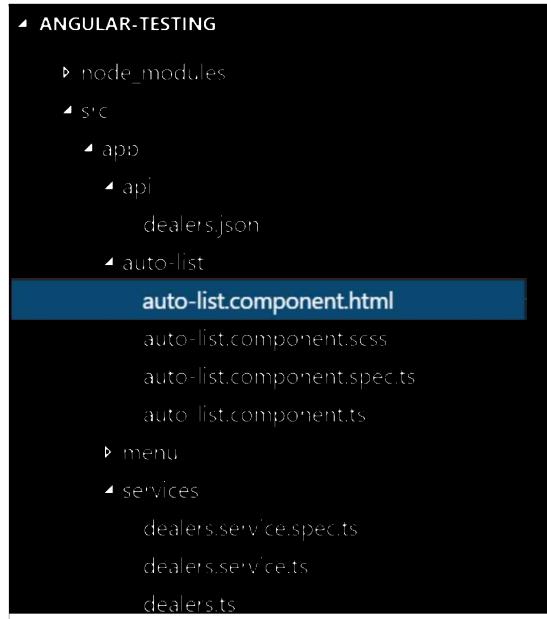
```

Eureka moment! The Angular CLI has been generating the required shell test scripts for the respective components and services. Let's do a quick hands-on exercise here. Let's generate a component named `auto-list`:

```
ng g component auto-list
```

The Angular CLI autogenerates the required files and also makes entries in the required files (`AppModule`, `Angular.json`, and so on).

The following screenshot depicts the test specs generated by the CLI:



Take a closer look at the files that were generated. You will see the following files generated for the component:

- `auto-list.component.html`
- `auto-list.component.spec.ts`
- `auto-list.component.ts`
- `auto-list.component.scss`

We are interested in the spec file generated by the Angular CLI. A spec file is the test script that was generated for the corresponding component. The spec file will have the basic required modules imported, along with the Component class. The spec file will also have some basic test specs already written, which can be used as a starting point or, alternatively, as our motivation.

Let's take a closer look at the code generated in the spec file:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { AutoListComponent } from './auto-list.component';
```

In the preceding code, you will notice that the required modules are imported from the Angular testing core. This is certainly not the final list of modules we will work with but just basic starter ones. You will also notice that the newly created component, AutoListComponent, is also imported into our spec file, which means that we can create an instance of our class inside the spec file and start mocking the objects for testing purposes. Pretty cool? Moving on to the lines of code, we can see the following:

```
describe('AutoListComponent', () => {
  let component: AutoListComponent;
  let fixture: ComponentFixture<AutoListComponent>;
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AutoListComponent ]
  })
  .compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(AutoListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

In the preceding code, you will notice some key points. There is a describe statement, which is used for grouping related test specs together. We will create test specs inside the describe function. There are two beforeEach methods defined in the spec file.

The first beforeEach method is an async promise, which will set up our TestBed, which means everything declared in it has to be resolved before moving on; otherwise, our tests won't work. The second beforeEach method will create an instance of our AutoList component for testing. You will notice the call to fixture.detectChanges(), which forces Angular's change detection to run and affect the elements in the test beforehand.

Now, it's time to understand the actual test spec, which is generated in the spec file:

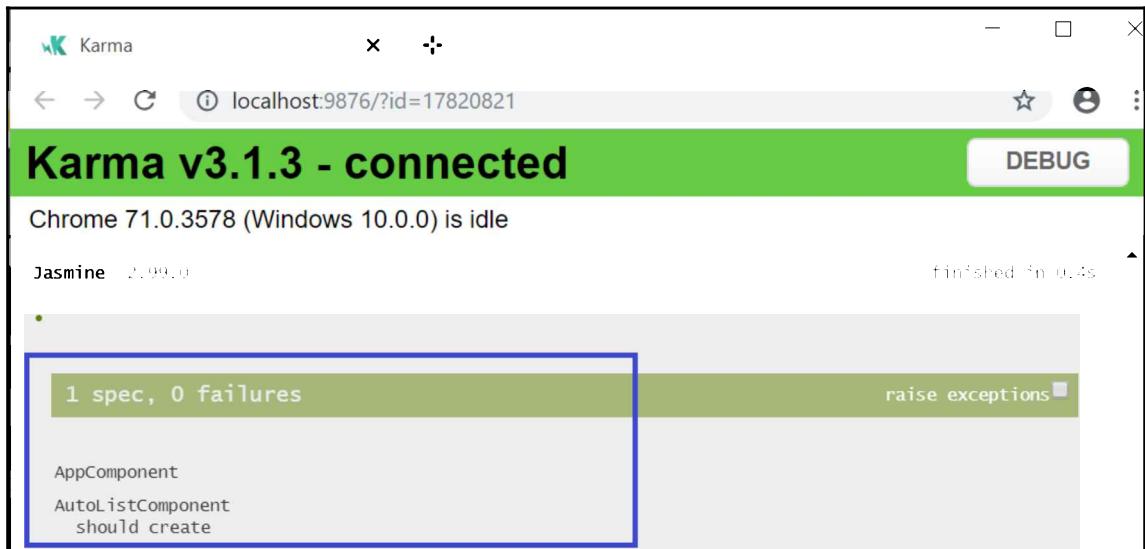
```
it('should create', () => {
  expect(component).toBeTruthy();
});
```

As we mentioned earlier, the Jasmine test specs are written inside the `it` statement, which, in this case, is just a simple assert to check whether the component exists and is true, using the `toBeTruthy` matcher.

That's all about our spec file. The joy lies in seeing it work. Let's just run the default tests that Angular has generated for us. To run the tests written inside the Angular application, we use the `ng test` command on the command-line interface:

```
ng test
```

If you see a new window being opened, don't panic. You will notice that a new browser window is opened by the Karma runner to execute the tests, and the test execution report is generated. The following screenshot displays the report that was generated for our test spec for the component:



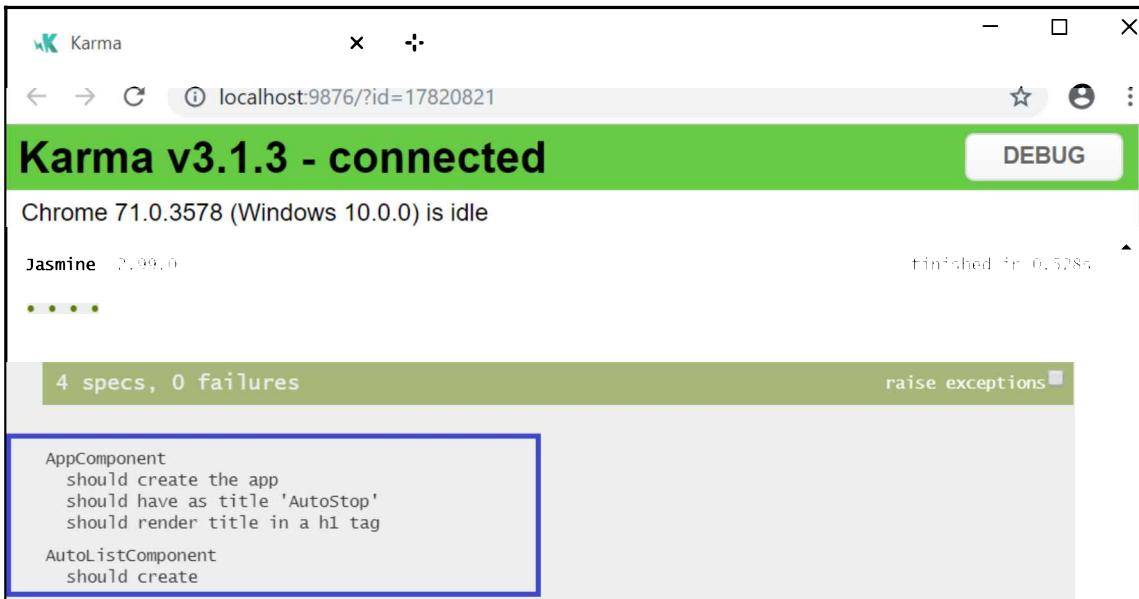
So, our test passed. Now, let's modify the script a bit. We will create a variable called `title` in our component and assign a value. In our test spec, we will verify whether the value matches or not. It's a straightforward use case and, trust me, it's also the most frequent use case you will implement in your applications. Let's open the `app.component.spec.ts` file and make the changes in the test script:

```
it(`should have as title 'testing-app'`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app.title).toEqual('AutoStop');
});
```

In the preceding code, we are writing a test spec and, using `TestBed`, we are creating a fixture element of `AppComponent`. Using the fixture element's `debugElement` interface, we are getting the `componentInstance` property. Next, we are writing an `expect` statement to assert if the value of the `title` variable is equal to `AutoStop`. That was neat. Let's try and write one more test spec. The use case we will address is as follows: we have an `H1` element and we want to assert it if the value inside the `H1` tag is equal to `Welcome to Autostop`. The following is the relevant sample code:

```
it('should render title in a h1 tag', () => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('h1').textContent).toContain('Welcome to
    AutoStop');
});
```

In the preceding code, we are asserting if the `textContent` of the `h1` element contains the text `Welcome to Autostop`. Notice that, in previous test specs, we used the `componentInstance` interface and that, in this test spec, we are using the `nativeElement` property. Again, run the tests using the `ng test` command. The following screenshot shows the test report that was generated:



So far, we have had an overview of the Jasmine and Karma frameworks, and also learned how to run our test scripts. We also learned about the default spec files that Angular generates for us and learned how to modify the test specs.

In the upcoming sections, we will learn how to write test specs and scripts to test Angular built-in directives, services, routes, and much more.

Testing directives

Angular provides a lot of built-in powerful directives, such as `ngFor`, `ngIf`, and so on, which can be used to extend the behavior and functionality of the native HTML elements. We learned about the Angular templates and directives in [Chapter 7, Templates, Directives, and Pipes](#). A quick recap has never hurt anyone. Angular offers us two types of directives that we can use to develop and extend the behavior of elements:

- Built-in directives
- Custom-defined directives

The focus of this section is to learn how to write test scripts for built-in Angular directives, such as `ngIf`, `ngFor`, `ngSwitch`, and `ngModel`. Before we start writing our test scripts, we need to do some groundwork to update our component so that we can start writing the test use cases. We will write a few variables, which will hold various types of data. We will display the data in our template using `ngFor` and also write some conditional checks using `ngIf`.



If you want a quick revision of Angular templates and directives, refer to Chapter 7, *Templates, Directives, and Pipes*.

We will continue to use the same component, `AutoListComponent`, which we created in the previous section. Let's start the party. Our starting point will be the `AutoListComponent` class, so let's modify the `auto-list.component.ts` file:

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

@Component({
  selector: 'app-auto-list',
  templateUrl: './auto-list.component.html',
  styleUrls: ['./auto-list.component.scss']
})
export class AutoListComponent implements OnInit {

  cars = [
    { 'id': '1', 'name': 'BMW' },
    { 'id': '2', 'name': 'Force Motors' },
    { 'id': '3', 'name': 'Audi' }
  ];

  tab = "1";

  constructor() { }

  ngOnInit() {
  }

  findAuto() {
    console.log("Method findAuto has been called");
  }
}
```

In the preceding code, we are adding a variable of a JSON object type called `cars` and assigning data to it. We will use this data by displaying it in the template. We are also declaring a variable, `tab`, and assigning a value, 1. We will use the `tab` variable for conditional checks in the template. Finally, we are adding a method, `findAuto`, and just displaying the output in the console.

We have modified our component class. We will also need to update our template file in order to process the data inside the component. The following is the sample code that we will add in our template file, `auto-list.component.html`:

```
<h4 class="c2">ngFor directive</h4>
<ul class="cars-list">
  <li *ngFor="let car of cars">
    <a [routerLink]=[car.id]">{{ car.name }}</a>
  </li>
</ul>

<h4 class="c1">ngIf directive</h4>
<div *ngIf="cars.length" id="carLength">
  <p>You have {{cars.length}} vehicles</p>
</div>

<h4 class="c3">ngSwitch directive</h4>
<div [ngSwitch]="tab" class="data-tab">
  <p>This is ngSwitch example</p>
  <div *ngSwitchCase="1">ngSwitch Case 1</div>
  <div *ngSwitchCase="2">ngSwitch Case 2</div>
</div>
<hr>

<button (click)="findAuto()" id="btn">Click to findAutoDealers</button>
```

In the preceding code, we are making the changes to the template file. First, we are using the `ngFor` directive to loop the rows and display the cars. Next, we are adding an `ngIf` condition to check whether the length of the car is more than 0, and then we will display the count of the `carLength` element. We have added an `ngSwitch` directive to check whether the value of the `tab` variable is set and, based on the value of the tab, we will display the respective tab, accordingly. In our case, since the value assigned to the `tab` is 1, we will display the first tab. Finally, we have added a button and associated the `findAuto` method with the click event.

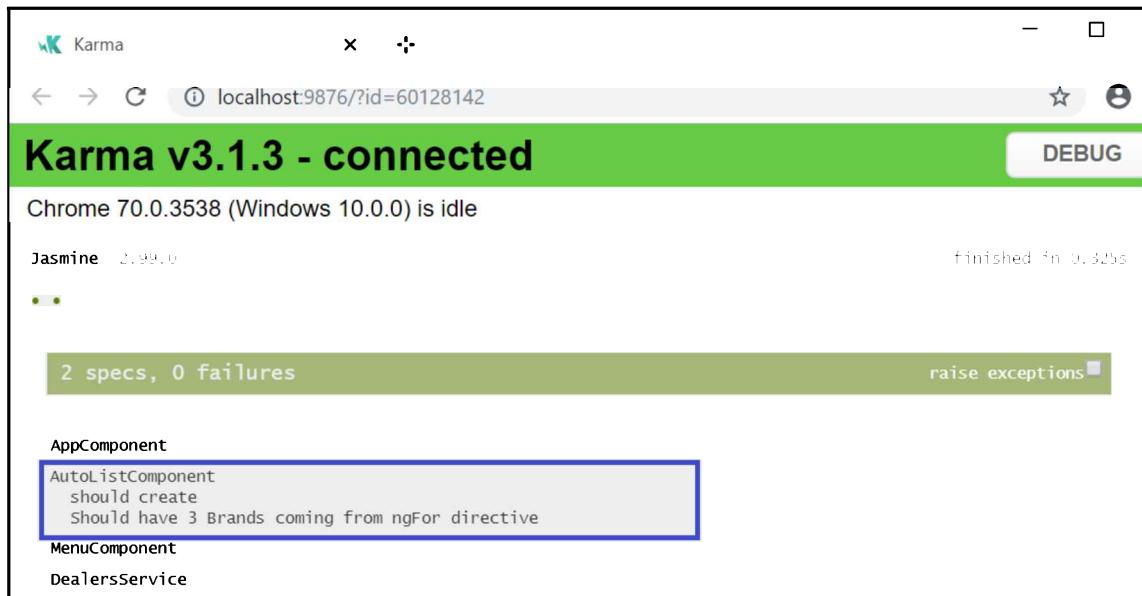
Beautiful. Our component and template are ready, and now it's time to write some good test scripts to test the preceding logic and, especially, the Angular built-in directives. Some of the use cases we will test include testing the count of cars displayed in the UI, testing which tab is active, verifying the content inside an element, and many more. Some of the use cases follow, and we will learn how to write test scripts for the use cases:

Use case #1: We have a list of cars and we want to verify that the total count is 3:

```
// ngFor test case to test the count is 4
it('Should have 3 Brands coming from ngFor directive', async(() => {
  const fixture = TestBed.createComponent(AutoListComponent);
  fixture.detectChanges();
  const el = fixture.debugElement.queryAll(By.css('.cars-list > li'));
  expect(el.length).toBe(3);
}));
```

In the preceding code, we are creating a fixture of the AutoListComponent component. We have already learned how to target an element using debugElement and, in this test spec, we are using the queryAll method to get the list of elements with className .cars-list > li. Finally, we are writing an expect statement to assert if the total count equals 3.

Run the tests using the `ng test` command. We should see the following output:

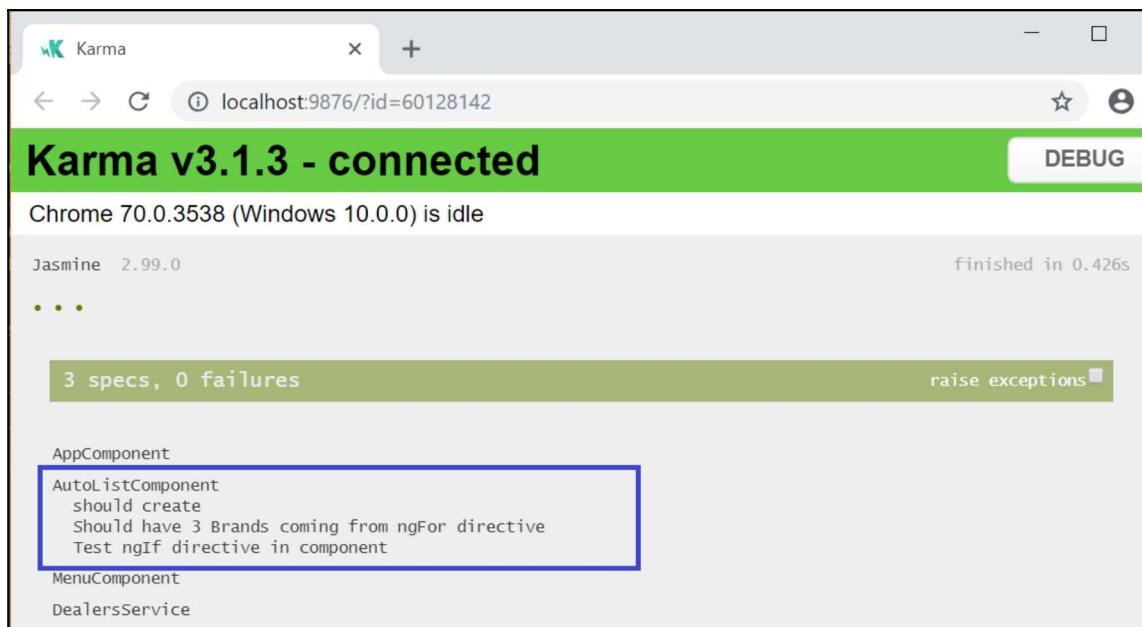


Use case #2: We want to verify that the text inside an HTML element contains the vehicles keyword:

```
// ngIf test script
it('Test ngIf directive in component', async(() => {
const fixture = TestBed.createComponent(AutoListComponent);
fixture.detectChanges();
const compiled = fixture.debugElement.nativeElement;
const el = compiled.querySelector('#carLength');
fixture.detectChanges();
const content = el.textContent;
expect(content).toContain('vehicles', 'vehicles');
}));
```

There are some important things to note in the preceding code. We continue to use the same fixture element of the component, AutoListComponent. This time, using the debugElement interface, we are using the querySelector method to find an element that has its identifier as carLength. Finally, we are writing an expect statement to assert if the text content contains the vehicles keyword.

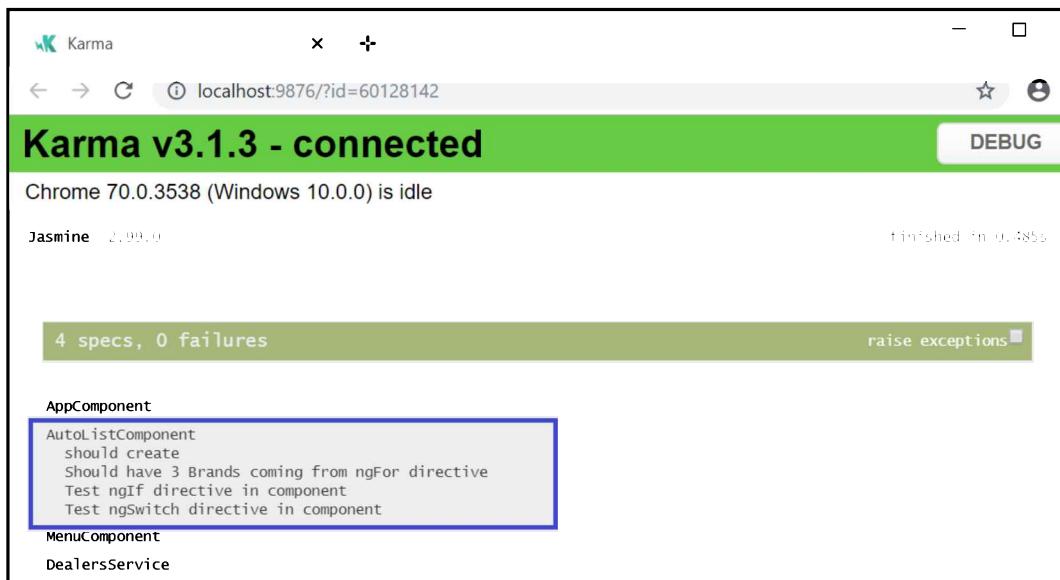
Let's run the tests again using the `ng test` command. We should see the following output:



Use case #3: We want to use ngSwitch to verify that tab1 is selected and, if so, display the corresponding div:

```
// ngSwitch test script
it('Test ngSwitch directive in component', async(() => {
const fixture = TestBed.createComponent(AutoListComponent);
fixture.detectChanges();
const compiled = fixture.debugElement.nativeElement;
const el = compiled.querySelector('.data-tab > div');
const content = el.textContent;
expect(content).toContain('ngSwitch Case 1');
}));
```

In the preceding code, we continue to use the fixture element of the AutoListComponent component. Using the debugElement and querySelector methods, we are targeting the element using className '.data-tab > div'. We are asserting whether the ngSwitch condition is true and the corresponding div is displayed. Since we have set the value of the tab to 1 in our component, tab1 is displayed on the screen and the test spec passes:



Use case #4: Test the methods defined inside AutoListComponent and assert whether the method has been called:

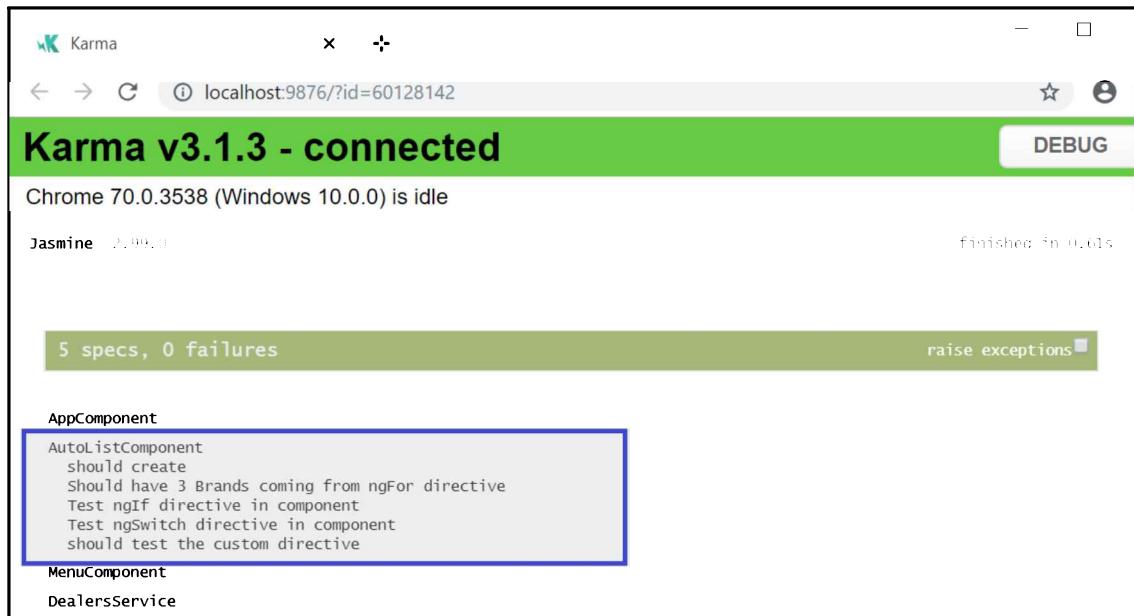
```
// Test button is clicked
it('should test the custom directive', async(() => {
```

```
const fixture = TestBed.createComponent(AutoListComponent);
component = fixture.componentInstance;
fixture.detectChanges();
spyOn(component, 'findAuto');
component.findAuto();
expect(component.findAuto).toHaveBeenCalled();

});
```

In the preceding code, we are creating a fixture of the `AutoListComponent` component. We are using the `spyOn` method to spy on the component instance. We are calling the `findAuto()` method. Finally, we are writing an `expect` statement to assert whether the `findAuto` method, using `toHaveBeenCalled`, has been called or not.

Run the tests using the `ng test` command. We should see the following output:



In this section, we learned how to write unit test scripts for testing Angular built-in directives, such as `ngFor`, `ngIf`, `ngSwitch`, and finally, asserting whether a method was clicked and called.

In the next section, we will learn about testing Angular routing.

Testing Angular routing

Most likely, you will have multiple links throughout the application in the form of a navigation menu or deep links. These links are treated as routes in Angular and are usually defined in your `app-routing.module.ts` file.

We learned about and mastered how to use Angular routing in [Chapter 4, Routing](#). In this section, we will learn how to write test scripts for testing Angular routing and testing the links and navigation in our application.

We will need a beautiful menu component for our application. Using the `ng generate component menu` command, we will generate the menu component. Now, let's navigate to `menu.component.html` and create a menu called navbar with two links in it:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">AutoStop </a>
  <button class="navbar-toggler" type="button" data-toggle="collapse"
    data-target="#navbarSupportedContent" aria-
    controls="navbarSupportedContent"
    aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-link" routerLink="/list-cars">Cars <span class="sr-only">
          (current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" routerLink="/list-trucks">Trucks</a>
      </li>
    </ul>
  </div>
</nav>
```

The preceding code is nothing fancy, at least not yet. It is standard code that uses Bootstrap to generate a navbar component. Look carefully and you will see that we have defined two links in the menu bar, `list-cars` and `list-trucks`, with the classes as `nav-link`.

We can now write a few test specs around the menu functionality to test the navbar component, which will cover navigation, the count of links, and so on.

Use case #1: We need to test that the navbar menu has exactly two links.

Here's the code to check whether there are exactly two links:

```
// Check the app has 2 links
it('should check routerlink', () => {
  const fixture = TestBed.createComponent(MenuComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;

  let linkDes = fixture.debugElement.queryAll(By.css('.nav-link'));
  expect(linkDes.length).toBe(2);

});
```

In the preceding code, we are creating a fixture for our `MenuComponent` component. Since we have assigned the `nav-link` class, it's easy to target the corresponding links in the component. Using the `debugElement` and `queryAll` methods, we are finding all the links with `className` as `nav-link`. Finally, we are writing an `expect` statement to assert whether the length of the array of links returned is equal to 2.

Run the tests using the `ng test` command. We should see the following output:



That's a good start to testing our menu functionality. Now that we know there are two links in our menu, the next use case we want to test is whether the first link is `list-cars`.

The following is the code to test whether the first link in the array of links is `list-cars`:

```
// Check the app has first link as "List Cars"
it('should check that the first link is list-cars ', () => {
  const fixture = TestBed.createComponent(MenuComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;

  let linkDes = fixture.debugElement.queryAll(By.css('.nav-link'));

  expect(linkDes[0].properties.href).toBe('/list-cars', '1st link should
    go to Dashboard');
});
```

In the preceding code, we are creating a fixture for our `MenuComponent` component. Using the `debugElement` and `queryAll` methods, we are finding all the links with `className` as `nav-link`. We will be getting all the links that have the class name as `nav-link`. There can be multiple links in the menu, but we are interested in reading the `href` property of the first element through `index [0]` and asserting whether the value matches `/list-cars`.

Again, run the `ng test` command. We should see our test report updated, as shown in the following screenshot:



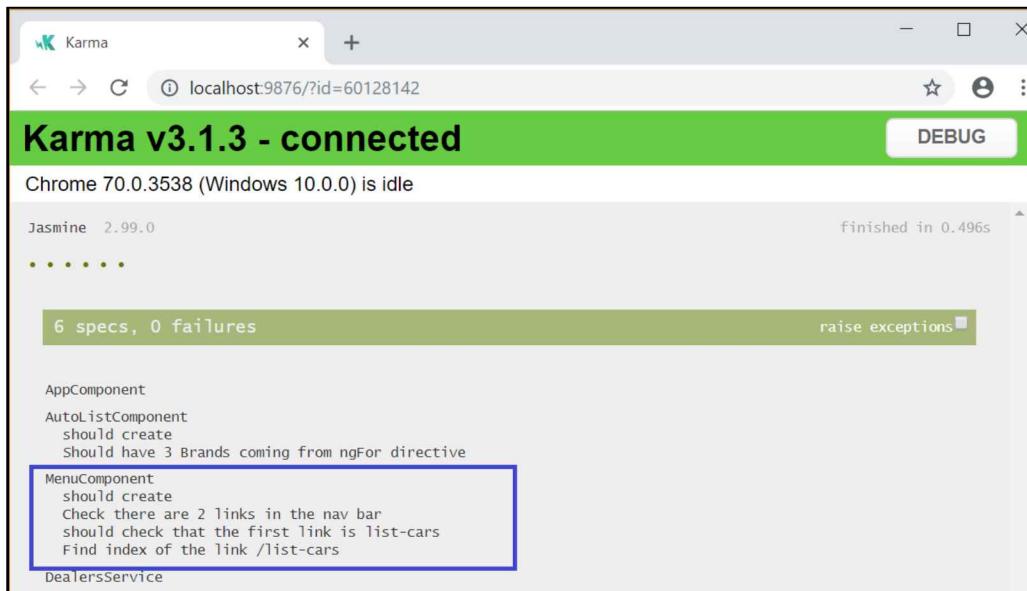
OK, fair enough. We got a clue that the `list-cars` menu link is the first in the menu list. What if we don't know the index or position of the link we are searching for? Let's tackle that use case as well.

Take a look at the following code snippet:

```
// Check the app if "List Cars" link exist
it('should have a link to /list-cars', () => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;
  let linkDes = fixture.debugElement.queryAll(By.css('.nav-link'));
  const index = linkDes.findIndex(de => {
    return de.properties['href'] === '/list-cars';
  });
  expect(index).toBeGreaterThan(-1);
});
```

Some things to note are that we are finding the index of the route path, `/list-cars`, and we are also making use of the assigned classes, `nav-link`, and getting an array of all matching elements using the `queryAll` method. Using the `findIndex` method, we are looping the array elements to find the index of the matching `href` to `/list-cars`.

Run the tests again using the `ng test` command and the updated test report should look as follows:



In this section, we learned about various ways to target a router link. The same principle applies to hunting down a deep link or a child link.

That's your homework.

Testing dependency injection

In the previous sections, we learned how to write test scripts for testing Angular components and routing. In this section, we will learn how to test dependency injection and how to test services in Angular applications. We will also learn how to inject services into Angular components and write test scripts to test them.

What is dependency injection?

Dependency injection (DI), in the Angular framework, is an important design pattern that allows the flexibility to inject services, interfaces, and objects into a class at runtime.

The DI pattern helps with writing efficient, flexible, and maintainable code that is testable and easy to extend.



If you need a quick recap, head over to [Chapter 11, Dependency Injection and Services](#), which covers and explains the DI mechanism in depth.

Testing Angular services

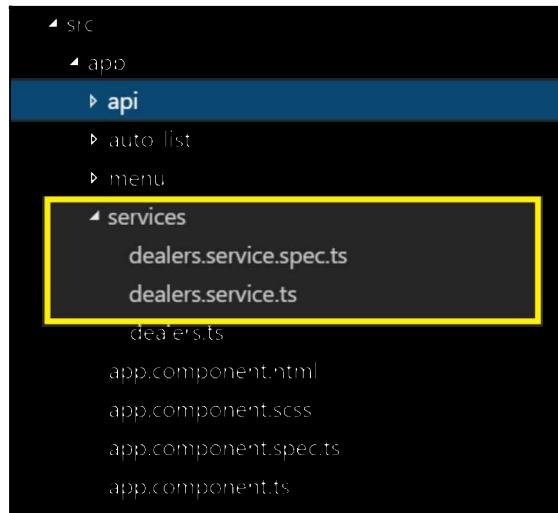
In this section, we will learn how to test Angular dependency injection through services and interfaces. In order to test an Angular service, we will first need to create a service in our app!

Use the `ng generate` command in the Angular CLI; we will generate the service in the project folder:

```
ng generate service services/dealers
```

Upon successful execution, we should see that the following files have been created:

- `services/dealers.service.spec.ts`
- `services/dealers.service.ts`



Now that we have our dealers service and the corresponding test spec file generated, we will work on our service to add a few methods and variables, so we will use them in our test specs. Navigate to our service class and update the `dealers.service.ts` file. The updated code should look as follows:

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

@Injectable({
  providedIn: 'root'
})
export class DealersService {
  dealers: any;

  constructor(private http : HttpClient) { }

  getDealers(){
    this.dealers = [
      { id: 1, name: 'North Auto'},
      { id: 2, name: 'South Auto'},
      { id: 3, name: 'East Auto'},
      { id: 4, name: 'West Auto'},
    ];
    return this.dealers;
  }
}
```

In the preceding code, we are making simple changes so that we can write a few test specs around the dealers service. We have defined a variable of the any type. We are defining a `getDealers` method, which will return a JSON response with an `id` and `name` key pair. Alright, now let's come up with some use cases to write our test scripts, such as getting the count of dealers, finding a matching dealer, and so on.

Use case #1: When the `getDealers` method is called, it should return the list of dealers, and the count should be equal to 4.

The following is the test spec for this:

```
it('Test Dependency Injection to get 4 dealers', () => {
  const service: DealersService = TestBed.get(DealersService);
  let dealers = service.getDealers();
  expect(dealers.length).toBe(4);
});
```

Use case #2: We want to check whether the first dealer name is North Auto.

The following is the test spec for this:

```
it('Test if the first Dealer is North Auto', () => {
  const service: DealersService = TestBed.get(DealersService);
  let dealers = service.getDealers();
  expect(dealers[0].name).toBe('North Auto');
});
```

Amazing! So far, so good. So, we have learned how to write test specs for our newly created dealers service. That's only one part of dependency injection. As part of dependency injection, we may need to inject additional required classes at runtime into the service.

Let's quickly create a class called `Dealers` and define two variables in it, namely `username` and `name`. Now, let's save this file as `dealers.ts`:

```
export class Dealers {

  constructor(
    public username: string = '',
    public name: string = ''
  ) {};

}
```

We will now include the newly created class in our dealers service and create a method to initialize the class and create an object to return some data:

```
getDealerObject()
{
    this.dealerObj= new Dealers('World','Auto');
    return this.dealerObj;
}
```

That brings us to our next use case to test.

Use case #3: Testing dependency injection via classes that have been injected into a service.

Have a look at the following code:

```
it('Test if the dealer returned from object is World Auto', () => {
  const service: DealersService = TestBed.get(DealersService);
  let dealerObj = service.getDealerObject();
  expect(dealerObj.name).toBe('Auto');
});
```

In the preceding code, we have created an instance of our service and invoked the `getDealerObject()` method. We are asserting whether the value returned matches the `name` property of the response to `Auto`.

We are calling the method defined in a service, which, internally, is dependent on the `Dealers` class.

Use case #4: What if we want to test just the properties of the `Dealers` class?

We can test that, too. The following is the sample code for this:

```
it('should return the correct properties', () => {
  var dealer = new Dealers();
  dealer.username = 'NorthWest';
  dealer.name = 'Auto';

  expect(dealer.username).toBe('NorthWest');
  expect(dealer.name).toBe('Auto');

});
```

Now, let's run the `ng test` command. We should see the following output:

The screenshot shows a browser window titled "Karma v3.1.3 - connected". The address bar says "localhost:9876/?id=60128142". The top right has a "DEBUG" button. Below it, "Chrome 70.0.3538 (Windows 10.0.0) is idle" and "Jasmine 2.99.0" are displayed. On the right, "finished in 0.51s" is shown. A green bar at the bottom left says "9 specs, 0 failures" with a "raise exceptions" checkbox. The main area lists test cases: "AppComponent", "AutoListComponent", "MenuComponent", and "DealersService". The "DealersService" section is highlighted with a blue border. At the bottom, there are tabs for "AutoStop", "Cars", and "Trucks".

On the same lines, you can write test scripts to test your services, dependency classes, or interface classes.

Use case #5: Testing Angular services inside a component.

We will continue to test Angular dependency injection. This time, we will import our services into the component and verify that it's working as expected.

In order to implement this use case, we will need to make changes to `AutoListComponent`.

Take a look at the changes we will make in the `auto-list.component.ts` file:

```
import { DealersService } from '../services/dealers.service';
constructor(private _dealersService : DealersService) { }
findAuto() {
  this.dealers = this._dealersService.getDealers();
  return this.dealers;
}
```

In the preceding code, we are importing the dealers service into the component. We are creating an instance of the service in the constructor method. We added a `findAuto` method, which calls the `getDealers` method using the instance of the class `_dealersService` service. In order to test the service in our component, let's modify the `auto-list.component.spec.ts` file by adding the following code:

```
import { DealersService } from '../services/dealers.service';
beforeEach(() => {
  fixture = TestBed.createComponent(AutoListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
  service = TestBed.get(DealersService);
});
```

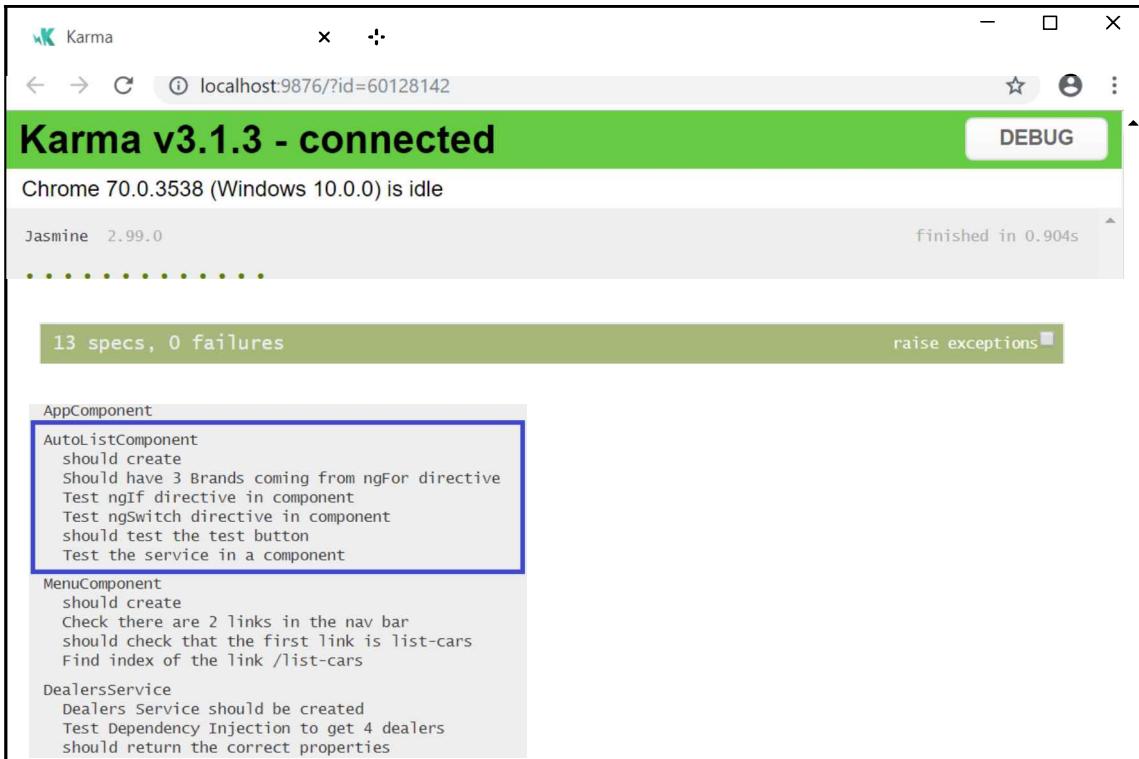
In the preceding code, we have imported our service `dealers` into the test spec file of `AutoListComponent`. We are creating an instance of the service using `TestBed` in the `beforeEach` method. We are now good to start writing our test specs in order to test the service. Add the following code to `auto-list.component.spec.ts`:

```
it('should click a button and call method findAuto', async(() => {
  const fixture = TestBed.createComponent(AutoListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
  spyOn(component, 'findAuto');
  let dealers = component.findAuto();
  expect(dealers.length).toEqual(4);

}));
```

In the preceding code, using the instance of the component, we are calling the `findAuto` method, which will return the data from the service. It expects the count to be equal to 4.

Run the tests using the `ng test` command. We should see the following output:



The screenshot shows a Karma test runner window titled "Karma v3.1.3 - connected". The browser tab is "localhost:9876/?id=60128142". The status bar indicates "Chrome 70.0.3538 (Windows 10.0.0) is idle". The test results show "Jasmine 2.99.0" and "finished in 0.904s". A green bar at the bottom displays "13 specs, 0 failures" and a "raise exceptions" button. The test results are listed under "AppComponent", which is highlighted with a blue border. The tests include:

- AutoListComponent
 - should create
 - Should have 3 Brands coming from ngFor directive
 - Test ngIf directive in component
 - Test ngSwitch directive in component
 - should test the test button
 - Test the service in a component
- MenuComponent
 - should create
 - Check there are 2 links in the nav bar
 - should check that the first link is list-cars
 - Find index of the link /list-cars
- DealersService
 - Dealers Service should be created
 - Test Dependency Injection to get 4 dealers
 - Should return the correct properties

In this section, we learned about various techniques to test Angular dependency injection, including services, dependency classes, and testing services inside Angular components.

Testing HTTP

In Chapter 12, *Integrating Backend Data Services*, we learned about integrating backend services and also learned about `HTTPModule` and `HttpClient`. We also learned how to make HTTP requests to the server and process the responses.

In this section, we will learn how to write test scripts to test HTTP requests and responses. We will continue to use the same project we created in this chapter—the AutoStop project. Before we proceed further, it's important to have the REST API endpoints ready so that we can use them in our application.

We will learn how to use the public API, <https://jsonplaceholder.typicode.com/>, which is available on the internet for free. We will also create a local server to return a mock JSON response from a local static JSON file.



We must import `HttpClientModule` and `HttpClientTestingModule` into our `app.module.ts` file.

Before we proceed to write our test scripts for testing Angular HTTP, we will need to update our dealers service, which we have used throughout this chapter. We will implement a few methods that will make HTTP calls—POST/GET to process data to the REST API endpoints.

We are working on the `dealers.service.ts` file, as follows:

```
import { HttpClient } from '@angular/common/http';
import { HttpHeaders, HttpParams, HttpResponse } from
  '@angular/common/http';
readonly REST_ENDPOINT = 'https://jsonplaceholder.typicode.com/users';
readonly DEALER_REST_ENDPOINT =
  'https://jsonplaceholder.typicode.com/users/1';
private _carurl = 'http://localhost:3000/cars';
```

In the preceding code, we are importing the required HTTP modules; that is, `HttpClient`, `HttpHeaders`, `HttpParams`, and `HttpResponse`, and also defining two REST endpoints that have the API URL for users and a specific user.



We can also have a local server up and running. You can have local APIs using the JSON server. You can learn more about this at <https://github.com/typicode/json-server>.

It's time to add a few methods, through which we will make the HTTP calls to the REST endpoints:

```
getAllDealers()
{
  this.allDealers = this.http.get(this.REST_ENDPOINT,
  {
    headers: new HttpHeaders().set('Accept', 'application/json')
  });
  return this.allDealers;
}

getDealerById(){
```

```
let params = new HttpParams().set('id', '1');
this.dealerDetails = this.http.get(this.REST_ENDPOINT, {params});
return this.dealerDetails;
}
```

In the preceding code, we are creating two methods, which make an HTTP GET request. The first method, `getAllDealers`, makes a call and expects a JSON response of users. The second method, `getDealerById`, will pass `id` as 1 and expect a single user data response. In the `getDealerById` method, we are using `HttpParams` to set the parameters to send to the endpoint. We will also modify our `autoListComponent` component to add a few methods to our Component class.

We are adding the following code to our `auto-list.component.ts` file:

```
findAuto() {
  this.dealers = this._dealersService.getDealers();
  return this.dealers;
}

listAllDealers(){
  this.allDealers = this._dealersService.getAllDealers();
}

listDealerById(){
  this.showDealerInfo = true;
  this.dealerDetail = this._dealersService.getDealerById();
  return this.dealerDetail;
}

getCarList() {
  this.carList = this.http.get<Cars[]>(this._carurl);
}
```

In the preceding code, we are adding a few methods, namely `findAuto`, `listDealerById`, and `getCarList`, which are making HTTP calls and calling methods that are in the `dealers` service.

Alright, now that we have our component and services set up, which are making HTTP calls, we are good to write our tests for HTTP.

Use case #1: We want to test whether a GET call was made to a particular URL.

We will add the following code to the `auto-list.component.spec.ts` file:

```
// Test HTTP Request From Component
it('Test HTTP Request Method', async(() => {
  const fixture = TestBed.createComponent(AutoListComponent);

  component = fixture.componentInstance;
  httpMock = TestBed.get(HttpTestingController);

  let carList = component.getCarList();

  fixture.detectChanges();
  const req = httpMock.expectOne('http://localhost:3000/cars');

  expect(req.request.method).toBe('GET');
  req.flush({});

}));
```

In the preceding code, we are creating the instance of `AutoListComponent`, using which we will make a call to its `getCarList` method. In the `getCarList` method, we are making a call to the `http://localhost:3000/cars` URL to retrieve data. We are creating an instance of the `HttpTestingController` class named `httpMock`. Using the `httpMock` instance, we are asserting that at least one call should be made to the URL.

Use case #2: We want to expect that the data returned as the result is more than 1:

```
it('Test HTTP Request GET Method With subscribe', async(() => {
  const fixture = TestBed.createComponent(AutoListComponent);
  component = fixture.componentInstance;
  component.listDealerById().subscribe(result =>
  expect(result.length).toBeGreaterThan(0));

}));
```

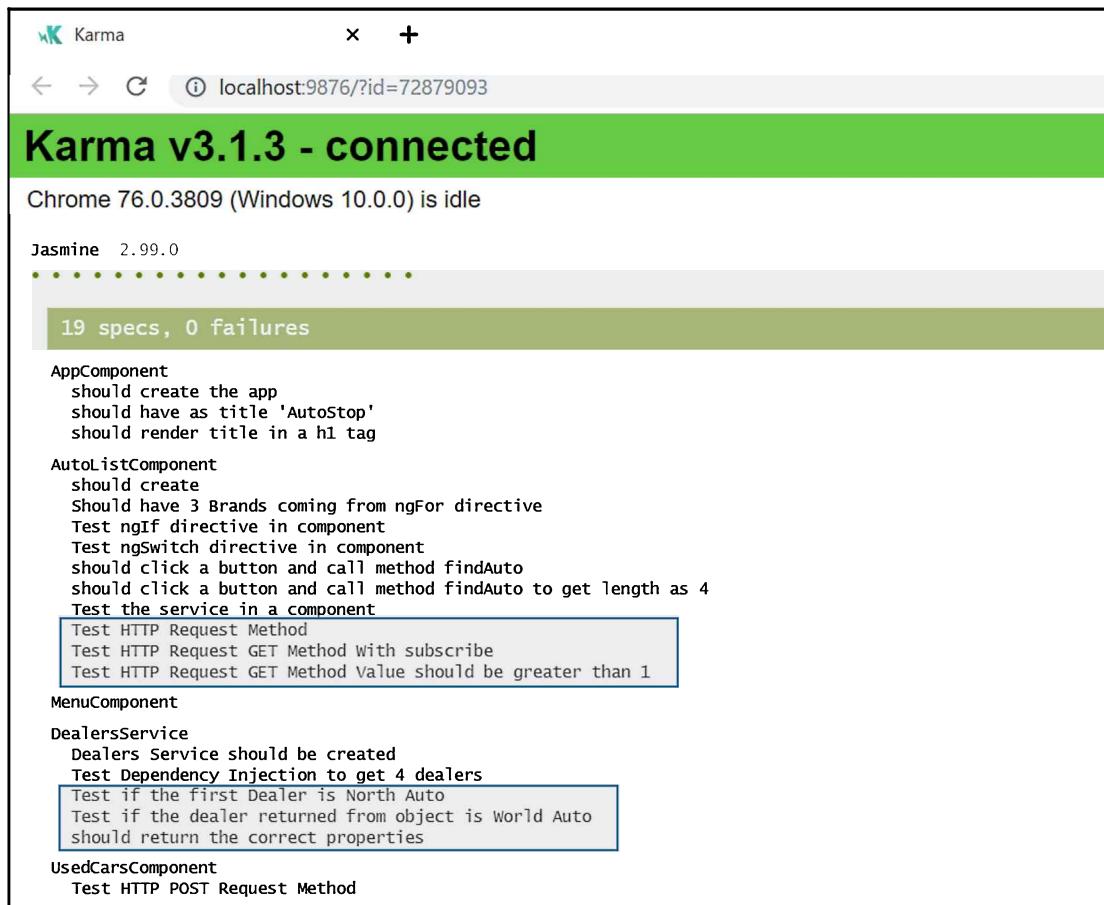
In the preceding code, using the instance of `AutoListComponent`, we are calling the `listDealerById` method. Using `subscribe`, we are mapping the result and verifying that the result data length is greater than 0.

Use case #3: We want to verify that the data returned from the HTTP call matches the data. The following is the sample code for this use case scenario.

```
it('Test if the first Dealer is North Auto', () => {
  const service: DealersService = TestBed.get(DealersService);
  let dealers = service.getDealers();
  expect(dealers[0].name).toBe('North Auto');
});
```

In the preceding code, using the `DealersService` instance, we are making a call to the `getDealers` methods. We are asserting data of the first index property name to be North Auto.

Run the tests using the `ng test` command. We should see the following output, as displayed and highlighted in the following screenshot:



If you see the preceding output, that's brilliant.

In this section, we have learned how to test components, services, and methods that are making HTTP request calls.

Summary

Testing is an important aspect of the application life cycle, and writing test scripts is crucial for application development success. We started with an overview of the frameworks supported by Angular, namely Jasmine and Karma. We learned how to run our tests using the `ng test` command. Then, we learned how to use the spec files autogenerated by Angular for all the components and services.

We learned how to write test scripts to test Angular components, built-in directives, services, and routing. We wrote test scripts for built-in directives, such as `ngFor`, `ngIf`, `ngSwitch`, and `ngModel`. We also covered use cases for testing Angular routing. Then, we created a `menu` component and wrote test scripts to test various use cases for the `menu` component.

We also explored testing dependency injection and services. We learned about various use cases and wrote test scripts for Angular services and HTTP calls.

In the next chapter, we will explore advanced Angular topics, such as custom directives and custom form validations.

Read on!

14

Advanced Angular Topics

In previous chapters, we learned how to use directives and form validators. We will extend our knowledge in this chapter with custom directives and custom validators. We're also going to look at how to build **single-page applications (SPAs)** with Angular.

Additionally, we'll explore integrating authentication into our Angular applications with two popular authentication providers: Google Firebase Authentication and Auth0.

This chapter will cover the following topics:

- Custom directives
- Custom form validators
- Building SPAs
- User authentication
- Authentication with Firebase Authentication
- Authentication with Auth0
- Wiring up the client side

Custom directives

In this section, we will learn how to create custom directives.

Firstly, let's understand what an Angular directive is.

Angular directives are a way to extend HTML functionality and the behavior of elements.

In previous chapters, we learned about and implemented many built-in directives, such as `*ngIf`, `*ngFor`, `*ngSwitch`, and `ngModel`.

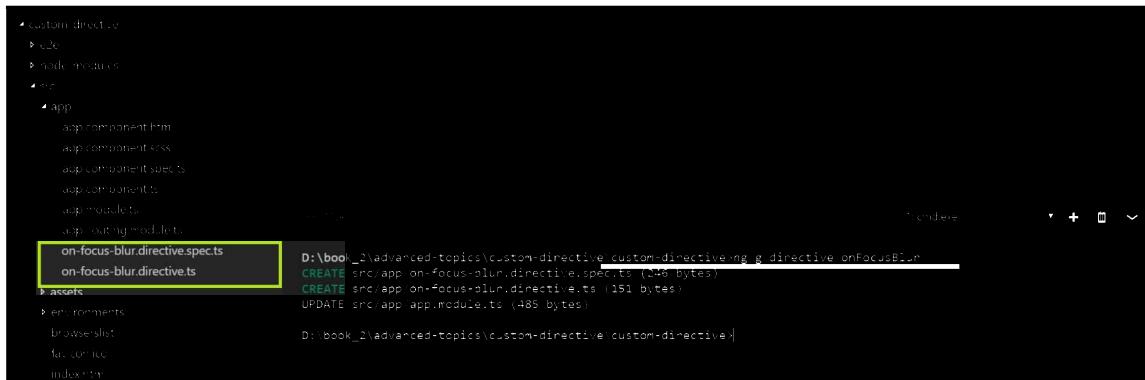
In this section, we will learn how to create our own custom directive to extend the functionality of HTML elements.

Use case: We want to create a custom directive for form elements and `onfocus`. The background color should be set to light blue, with the border dark blue, and the `onblur` event should be highlighted in red. So, let's begin:

1. Let's generate the directive using the `ng` command:

```
ng g directive onFocusBlur
```

On running the previous command, this is what will show up on our screen:



Notice that the directive files have been generated and that our `app.module.ts` file has also been updated, which means the directive is available across the app, to be used anywhere in any component.

2. In the directive file, `on-focus-blur.directive.ts`, add the following lines of code:

```
import { Directive } from '@angular/core';
import { HostListener, HostBinding } from '@angular/core';

@Directive({
  selector: '[appOnFocusBlur]'
})
export class OnFocusBlurDirective {
  constructor() { }
  @HostBinding("style.background-color") backgroundColor;

  @HostListener('focus') onFocus() {
    this.backgroundColor = '#19ffe4';
  }

  @HostListener('blur') onBlur() {
```

```
        this.backgroundColor = '#ff1934';
    }
}
```

In the preceding code, the following important things should be noted:

- We are importing the required modules, that is, Directive, HostListener, and HostBinding.
- Using the `@directive` decorator, we are defining the name of the directive through the selector.
- `@HostBinding` is used to set properties on the element.
- `@HostListener` is used to listen to the events on the host element.
- We are binding the style background color property in the preceding example. We can bind any style, class, or event property on the host element.
- Using `@HostListener`, we are listening to the events and, with `onFocus`, we are changing the background color. By using `onBlur`, we reset the color.

Now, we are good to use this decorator anywhere in our application.

3. We are going to use this in our `app.component.html` file on a form control input element:

```
<input type="text" appOnFocusBlur class="nav-search" >
```

4. Run the app using the `ng serve` command and click on the Input button. We should see the output and behavior, as shown in the following screenshot:



Great. Now that we know how to write our custom directives, we will go ahead and try creating our own custom directives.

In the next section, we will learn about writing custom form validations.

Custom form validations

In previous chapters, we learned about forms and implementing form validations. We used the built-in form validations or HTML5 attribute validations. But, in more complex scenarios, we will need to implement custom form validations. These validations differ from application to application. In this section, we will learn about custom form validations. To recap quickly, Angular provides us with various options through which we can implement form validations using the `Validators` module in Angular forms.

An example of using validators is shown in the following code:

```
loginForm = new FormGroup({
  firstName: new FormControl('', [Validators.required,
    Validators.maxLength(15)]),
  lastName: new FormControl('', [Validators.required]),
});
```

In the preceding code, using the `Validators` module, we are applying validations of `required`, `maxLength`, and so on.

Now, let's learn how to create our own custom form validations. First, we will generate a component in which we will implement a form and a few elements so that we can apply our newly created directive:

```
ng g c customFormValidation
```

Upon running the preceding command successfully, we should see the following output:

```
TERMINAL
chunk {main} main.js, main.js.map  (main) 25.5 KB [initial] [rendered]
i ｢wdm｣: Compiled successfully.
?C
D:\book_2\book\chapter15\user-authentication-firebase\angular-with-firebase\angular-firebase-app\ng g c customFormValidation
CREATE src/app/custom-form-validation/custom-form-validation.component.html (41 bytes)
CREATE src/app/custom-form-validation/custom-form-validation.component.spec.ts (728 bytes)
CREATE src/app/custom-form-validation/custom-form-validation.component.ts (832 bytes)
CREATE src/app/custom-form-validation/custom-form-validation.component.scss (0 bytes)
UPDATE src/app/app.module.ts (1845 bytes)
```

Now that we have generated our component, let's generate a directive in which we will implement custom form validations.

We will implement a custom directive to check the ISBN field.

What is an ISBN? An ISBN is a unique identifier for each book that is ever published.

Here are the conditions that are required for an ISBN number:

- The ISBN number should be exactly 16 characters
- Only integers are allowed for ISBNs

Now, using the ng command, we will generate our directive:

```
ng g directive validISBN
```

Upon successful execution of the above command we should see the output as shown in the screenshot below

```
D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase\angular-firebase-app>ng g d
irective validISBN
CREATE src/app/valid-isbn.directive.spec.ts (237 bytes)
CREATE src/app/valid-isbn.directive.ts (147 bytes)
UPDATE src/app/app.module.ts (1930 bytes)
```

In the `valid-isbn.directive.ts` file, add the following lines of code:

```
import { Directive } from '@angular/core';
import { NG_VALIDATORS, ValidationErrors, Validator, FormControl } from
  '@angular/forms';

@Directive({
  selector: '[validISBN]',
  providers: [
    { provide: NG_VALIDATORS,
      useExisting: ValidISBNDirective, multi: true }
  ]
})

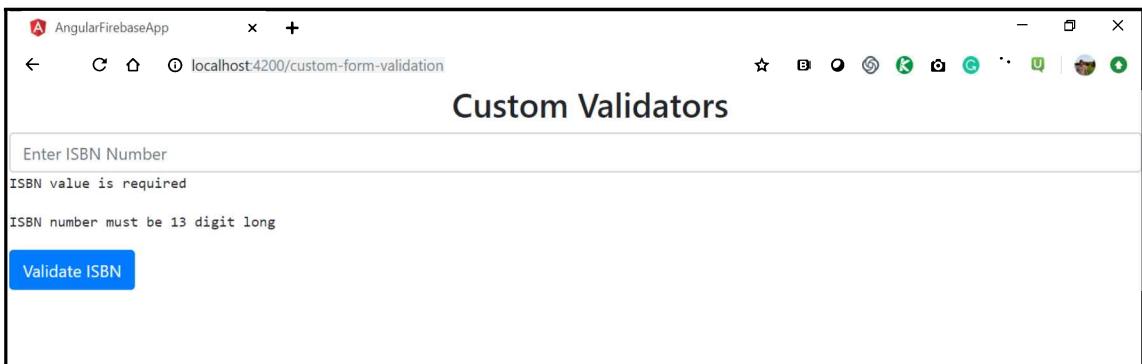
export class ValidISBNDirective implements Validator {
  static validateISBN(control: FormControl): ValidationErrors | null {
    if (control.value.length < 13) {
      return { isbn: 'ISBN number must be 13 digit long' };
    }
    if (!control.value.startsWith('Packt')) {
      return { isbn: 'Value should start with Packt' };
    }
  }
}
```

```
        return null;
    }

    validate(c: FormControl): ValidationErrors | null {
        return ValidISBNDirective.validateISBN(c);
    }
}
```

Let's analyze the preceding code snippet in detail. First, using the ng CLI commands, we have generated a directive named `validISBN`. The Angular CLI will autogenerate the required file, with the basic syntax prepopulated. We are importing the required modules, namely `NG_VALIDATORS`, `ValidationErrors`, `Validator`, and `FormControl`. We are injecting the required modules as part of our providers. Next up, we have implemented a method named `validateISBN`, which is taking a parameter of the `FormControl` type. We are passing our form control field to this method, which will validate whether the value of the form control matches the conditions implemented in the method. Finally, we are invoking the `validateISBN` method in the method `validate`.

Now, we are good to use this custom form validation in any number of places, that is, wherever we need to verify or validate the ISBN number. Let's run the application using the `ng serve` command. We should see the following output:



So far in this chapter, we have been applying some of them out of box, thinking and learning about how to build our custom directives and custom form validations. We have also learned how easy it is to integrate them into existing, or any new, applications effortlessly. All this can also form parts of single-page applications. Wait. What? Single-page applications? What's that? In the next section, we are going to learn all about single-page applications and build our own.

Building single-page applications

In this section, we will learn about building single-page applications.

What exactly is a single-page application?

A single-page application is a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entirely new pages from a server.

Think of it as an application with only one HTML file, and the contents of the page load dynamically based on the request made by the user. We only create templates that get rendered in our browser dynamically at runtime.

Let me give you a good example.

In Chapter 15, *Deploying Angular Applications*, using the `ng build` command, we generated the compiled code of an Angular app.

Take a look at the compiled source code that was generated by Angular:

| Name | Date modified | Type | Size |
|--------------------------------|--------------------|--------------------|--------|
| 3rdpartylicenses | 11/28/2018 2:19 PM | Text Document | 21 KB |
| favicon | 11/28/2018 2:19 PM | Icon | 6 KB |
| index | 11/28/2018 2:19 PM | Chrome HTML Doc... | 1 KB |
| main.3e61700c591a5c6168fa | 11/28/2018 2:19 PM | JavaScript File | 262 KB |
| polyfills.20ab2d163684112c2aba | 11/28/2018 2:19 PM | JavaScript File | 38 KB |
| runtime.ec2944dd8b20ec099bf3 | 11/28/2018 2:19 PM | JavaScript File | 2 KB |
| styles.3ff695c00d717f2d2a11 | 11/28/2018 2:19 PM | CSS Source File | 0 KB |

In the preceding screenshot, you will see only one HTML file, named `index`.

Go ahead and open the file—you will see it's blank. That's because Angular applications are single-page apps, which means the content and data will be generated on the fly dynamically based on user actions.



It's safe to say that all Angular applications are single-page applications.

The following are some of the advantages of building a single-page application:

- The pages are rendered dynamically, and therefore our application source code is secure.
- As the compiled source code renders in the user's browser, the pages load much faster than in the traditional request and response model.
- Since pages load faster, this leads to a better user experience.
- Using the `Router` component, we only load components and modules that are needed for certain features and do not load all of the modules and components in one go.

Throughout the course of this book, we have created many Angular apps, and each one of them has been a single-page application.

User authentication

In this section, we will learn how to implement user authentication in our Angular applications.

User authentication, in a broad context, consists of safely logging the user into our application, who should be able to view, edit, and create data on secure pages, and finally, log out of the application!

In a real-world application, there will be a lot of additional checks and security implementations to be done to sanitize user inputs, as well as checking whether they're a valid user, or verifying the authentication token for session timeouts, and other data checks to make sure no bad elements creep into the app.

The following are some important modules for user authentication:

- Signing up new users
- Login for existing users
- Password reset
- Session management for logged-in users
- One-time password or dual authentication
- Logging out an already logged in user

In the upcoming sections, we will learn about implementing the preceding functionality using the Firebase and Auth0 frameworks.

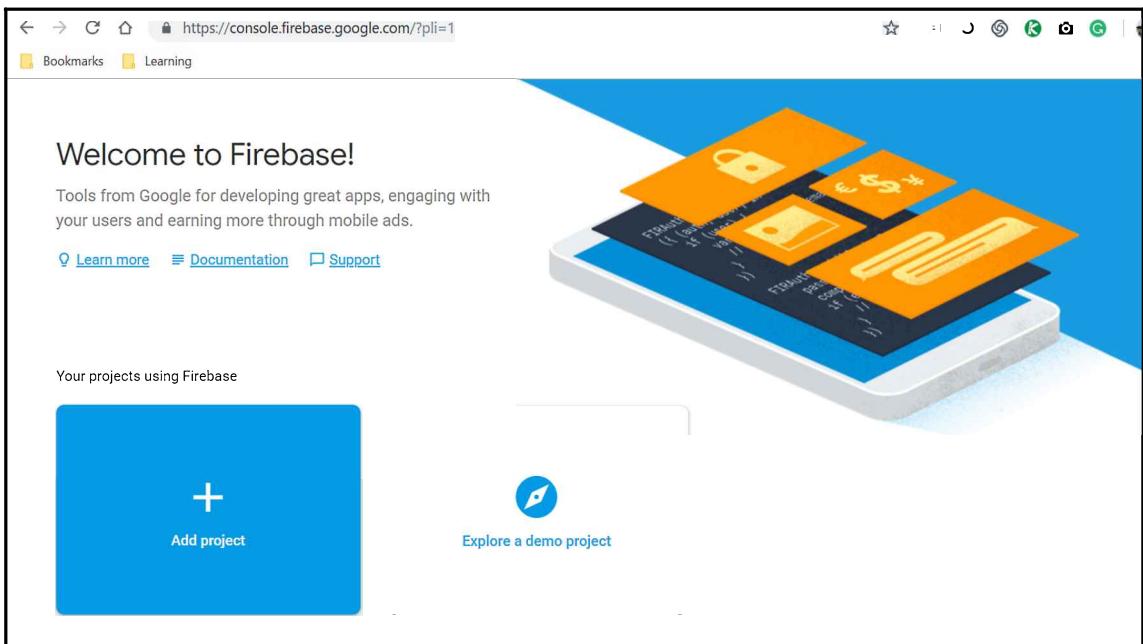
User authentication with Firebase

In this section, we will learn how to implement user authentication using Firebase.

What is Firebase?

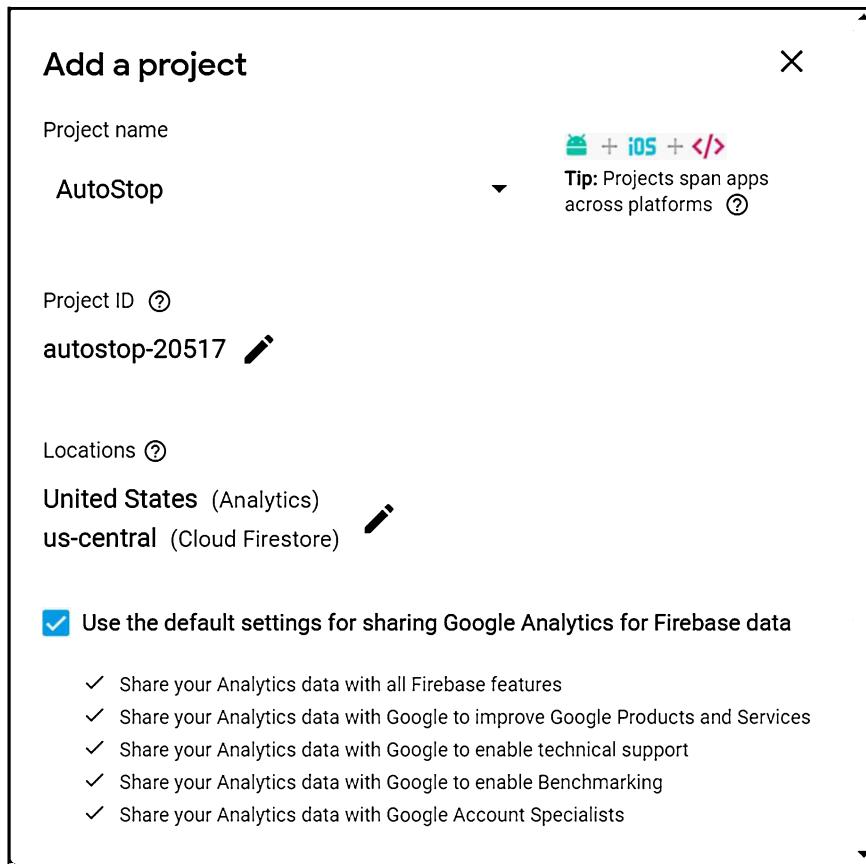
Firebase is a managed service provided by Google. Firebase gives us functionality such as analytics, databases, messaging, and crash reporting, so that we can move quickly and focus on our users. You can learn more about the service at <https://firebase.com>. Now, let's jump right in and implement Firebase in our Angular app.

The first step is to create an account with Google to use the Firebase service. You can use your Google account to log in to Firebase. Once you have successfully created your Firebase account, you should see the following output:



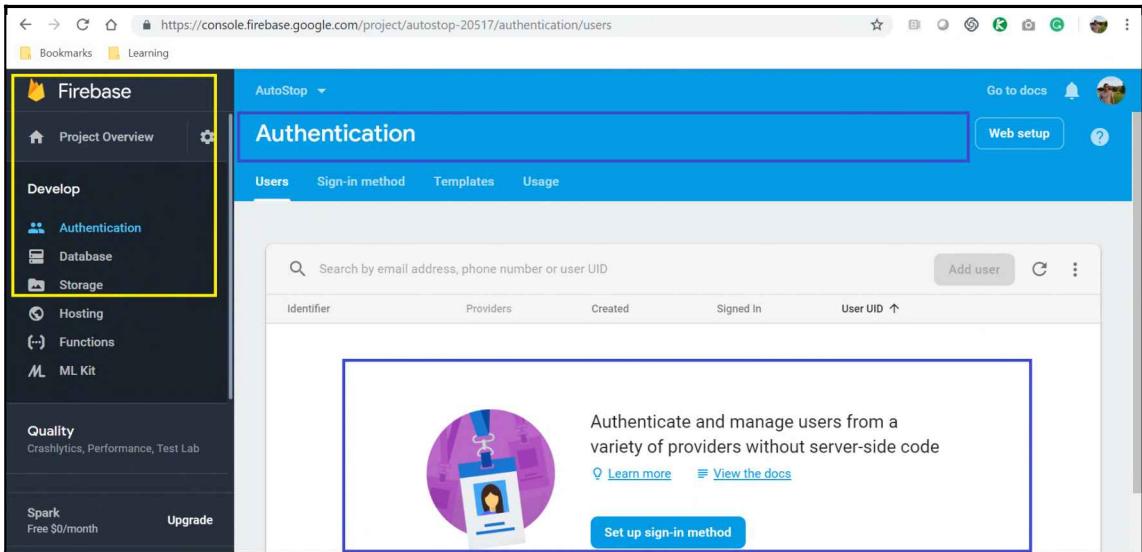
To create a new project, click on the **Add Project** link.

You will see the following dialog window, prompting you to enter the project's name; in our case, we are making our project name **AutoStop**:



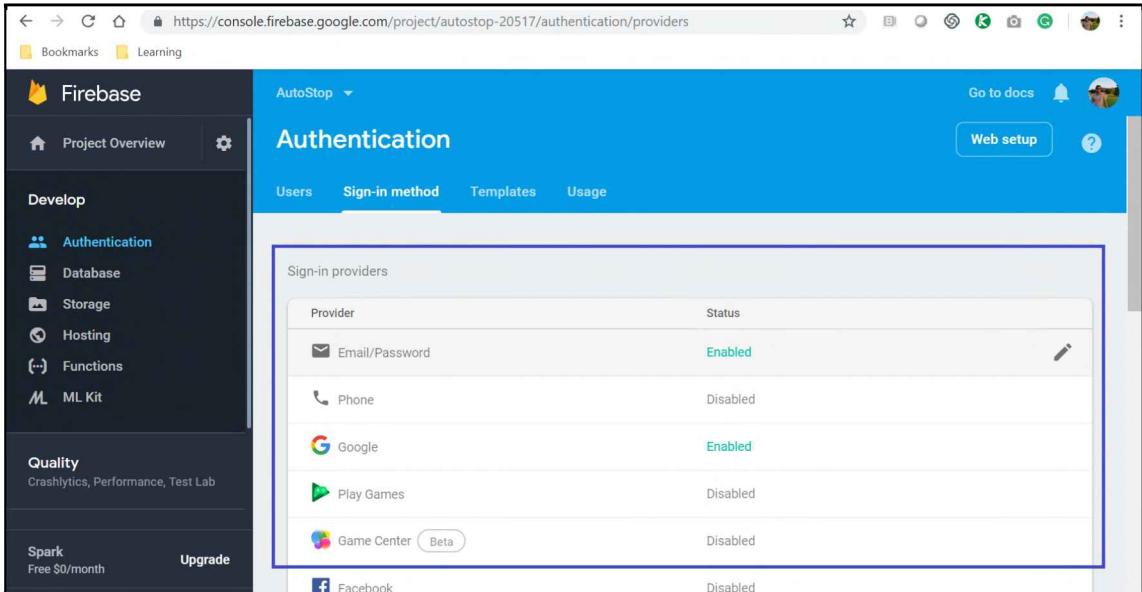
Note that Google will assign a unique project ID to your project.

Now, click on the **Authentication** link on the left-hand-side menu to set up user authentication features, which we can embed and set up in our Angular application:



We can do a lot of other cool stuff here, but we will focus on the **Authentication** module for now.

Now, click on the **Sign-in method** tab to set up options for how to allow users to sign in to our Angular application:

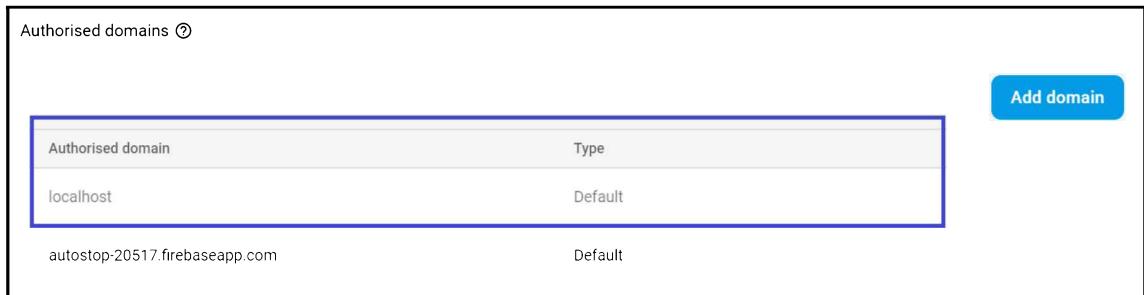


In the preceding screenshot, you will notice the following important things:

- Google Firebase provides various options that we can enable, through which we would want users of our application to sign in.
- We need to enable each provider option individually .
- We have enabled **Email/Password** and Google options in our application.
- In order to enable Facebook, Twitter, and other apps, we will need to enter the developer API keys provided by the respective services.

Now, scroll down a little bit on the page and you will see an option to set up called **Authorised Domains**.

We will see two default values set up, that is, localhost and a unique subdomain, on the Firebase application, as shown in the following screenshot:



| Authorised domain | Type |
|----------------------------------|---------|
| localhost | Default |
| autostop-20517.firebaseioapp.com | Default |

Add domain

We have made the required changes. Now, we need to set up Google Firebase's app settings. It's time to implement the user authentication in our Angular application.

Prerequisite: We expect users to have an Angular application up and running.

Open the Angular CLI command prompt; we need to install a few modules. We will need to install Angular Fire2 and Firebase first:

Please note that Angular Fire2 is now Angular Fire.



We will need to run the following command to install Angular Fire in our application:

```
npm install angularfire2
```

Upon successful execution of the preceding command, we should see the output shown in the following screenshot:

```
D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase> npm install angularfire2
npm WARN deprecated angularfire2@5.1.1: AngularFire has moved, we're now @angular/fire
npm WARN enoent ENOENT: no such file or directory, open 'D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase\package.json'
npm WARN enoent ENOENT: no such file or directory, open 'D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase\package.json'
npm WARN @angular/fire@5.1.1 requires a peer of @angular/common@>=6.0.0 <8 but none is installed. You must install peer dependencies yourself.
npm WARN @angular/fire@5.1.1 requires a peer of @angular/core@>=6.0.0 <8 but none is installed. You must install peer dependencies yourself.
npm WARN @angular/fire@5.1.1 requires a peer of @angular/platform-browser@>=6.0.0 <8 but none is installed. You must install peer dependencies yourself.
npm WARN @angular/fire@5.1.1 requires a peer of @angular/platform-browser-dynamic@>=6.0.0 <8 but none is installed. You must install peer dependencies yourself.
npm WARN @angular/fire@5.1.1 requires a peer of firebase@^5.5.0 but none is installed. You must install peer dependencies yourself.
npm WARN @angular/fire@5.1.1 requires a peer of zone.js@^0.8.0 but none is installed. You must install peer dependencies yourself.
npm WARN angular-with-firebase No description
npm WARN angular-with-firebase No repository field.
```

All set. Now, we need to create a service that will handle our authentication functionality:

```
ng g service appAuth
```

Using the ng command, we are generating a new service, named appAuth:

```
D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase\angular-firebase-app>ng g service appAuth
CREATE src/app/app-auth.service.spec.ts (339 bytes)
CREATE src/app/app-auth.service.ts (136 bytes)

D:\book_2\book\chapter14\user-authentication-firebase\angular-with-firebase\angular-firebase-app>
```

Now, it's time to modify the `appAuth.service.ts` file and add the following code to it:

```
import { Injectable } from '@angular/core';
import { AngularFireAuth } from '@angular/fire/auth';
import { auth } from 'firebase/app';
import { Router } from '@angular/router';

@Injectable({
providedIn: 'root'
})
export class AppAuthService {

    private authUser:any;
    private authState:any;
    private loggedInUser = false;
    private userToken = '';
```

```
constructor(public afAuth: AngularFireAuth, private router :Router) { }

login() {
  this.afAuth.auth.signInWithPopup(new auth.GoogleAuthProvider());
}

this.loggedInUser = true;

this.afAuth.currentUser.getIdToken(true).then(token => this.userToken =
token);

this.afAuth.authState.subscribe((auth) => {
  this.authState = auth;
});

this.router.navigate(['/profile']);
}

isLoggedInUser(){
if(this.userToken != '')
  return true;
else
  return false;
}

logout() {
  this.afAuth.auth.signOut();
  this.loggedInUser = false;
  this.userToken = '';
}

}
```

In the preceding code, we are making changes to the `app-auth.service.ts` file. The following important points should be noted:

- We are importing the required classes, namely `AngularFireAuth`, `Auth`, and `Router`, into the service.
- Using `@Injectable`, we are specifying that the service is injected at the root level in the Angular tree structure.
- We are defining a few private variables that we will use across our application.
- In the constructor method, we are injecting the `AngularFireAuth` and `Router` classes.
- We are defining three methods: `Login`, `Logout`, and `isLoggedInUser`.

- In the `login` method, we are using the `this.afAuth` instance, calling the `signInWithPopup` method, and passing the `auth.GoogleAuthProvider` argument, which we get from the Firebase app that we installed locally:

```
    this.afAuth.auth.signInWithPopup(new auth.GoogleAuthProvider());
```

- When this method is invoked, a new window will open up, in which we can see the Google sign-in option, using which we can log in to the app.
- We are setting the `this.loggedInUser` variable to `true`.
- We are setting the logged-in user's token to the `this.userToken` variable.
- We are also subscribing to get the `authState` response.
- Finally, using the router instance and using the `navigate` method, we are redirecting the user to the profile page.
- Inside the `isLoggedInUser` method, we are verifying whether the `userToken` is set or not. `userToken` will be set if the user has logged in correctly; otherwise, the method will return `false`.
- In the `logout` method, again using the instance of `afauth`, we are calling the `signout` method, which will log the user out.
- Finally, we are setting the `userToken` to empty.

Awesome. We have done all the heavy lifting in our `app-auth.service.ts` file. Now, it's time to call these methods in our components: `login`, `profile`, and `log out`.

In the `login.component.html` file, we will add the following login form:

```
<div *ngIf="!_appAuthService.loggedInUser">
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

  <button>Login</button>

</form>
</div>
```

In the preceding code, we are just adding an Angular reactive login form using `FormGroup` and `FormControllers`.

The output of the login form is shown in the following screenshot:



And in the `profile.component.ts` file, we are just making a call to the `login` method:

```
onSubmit() {
  this._appAuthService.login();
  console.warn(this.loginForm.value);
}
```

Now, in the `profile.component.ts` file, we add a check to see whether the user is logged in or not:

```
<div *ngIf="_appAuthService.isLoggedInUser">
<p>
  profile works!
</p>

User Token is {{_appAuthService.userToken}}
</div>
```

When the user navigates to the profile page, if they are logged in, they will see the details; otherwise, the user will be redirected to the login page.

Now, on to the final part; we will have a logout link in our `app.component.html` file:

```
<nav>
  <a routerLink='/login' *ngIf="!_appAuthService.isLoggedIn()">Login</a>
  <a routerLink='/register'>Register</a>
  <a routerLink='/logout'
    *ngIf="_appAuthService.isLoggedIn()">Logout</a>
</nav>
```

We are adding links with `*ngIf` conditions to show the corresponding links when the user is logged in or not:

```
ngOnInit() {
  this._appAuthService.logout();
  this.router.navigate(['/login']);
}
```

When the user clicks on the logout link, we are calling the `logout` method of `appAuthService` and, on successful logout, we are redirecting the user back to the login page.

Now, let's run the app using the `ng serve` command. We should see the following output:



User authentication with Auth0

In this section, we will learn how to implement user authentication using **Auth0**. Before we go ahead and implement **Auth0** in our Angular application, we will need to implement some prerequisites. Let's get right to it:

1. First, we will need to create an account with **Auth0** at [Auth0.com](https://auth0.com). Upon successfully logging in to the account, we should see the following dashboard screen:

The screenshot shows the Auth0 dashboard at <https://manage.auth0.com/#/>. At the top, there is a message about a 18-day trial left for the Free plan, with a link to provide billing information. Below the message, there is a 'BILLING' button. The main area is titled 'Dashboard' and includes a sidebar with links to 'Dashboard', 'Applications', 'APIs', 'SSO Integrations', 'Connections', 'Users', 'Rules', 'Hooks', 'Multi-factor Auth', and 'Hosted Pages'. On the right, there is a 'Login Activity' chart showing activity from January to December. Below the chart, there are three summary metrics: 'USERS' (0), 'LOGINS' (0), and 'NEW SIGNUPS' (0). A 'Continue with this tutorial' button is also present.

We will have to register our application so that we can create the required settings to implement Auth0 in our app.

2. Click on the **Applications** link on the left-hand-side menu:

The screenshot shows the Auth0 Applications dashboard at <https://manage.auth0.com/#/applications>. The left sidebar includes links for Dashboard, Applications (which is highlighted in red), APIs, SSO Integrations, Connections, Users, Rules, Hooks, Multi-factor Auth, and Hosted Pages. The main area displays a single application named "Default App" under the "GENERIC" category. The application's Client ID is listed as "USSLa5ZybkVTy4PJ-KQSTE-KnhTPt". A "TUTORIAL" button is visible above the applications list, and a "+ CREATE APPLICATION" button is in the top right corner.

3. Now, click on the **Create Application** button to create an application:

The screenshot shows the "Create Application" form at <https://manage.auth0.com/#/applications>. The "Name" field contains "AutoStop". The "Choose an application type" section offers four options: "Native App", "Single Page Web App" (which is selected and highlighted in blue), "Regular Web App", and "Machine to Machine App". The "Single Page Web App" description states it's a "JavaScript front-end app that uses an API" with examples like "AngularJS + NodeJS". A "CREATE" button is at the bottom, and a "Continue with this tutorial" link is in the bottom right.

4. We will need to enter the name of the application and select the type of application we are building. In our case, it's a **Single Page Web App**, so go ahead and select the option and click on the **CREATE** button.
5. The next thing we need to do is update the important settings of our application. So, click on the application name and navigate to the **Settings** tab:

The screenshot shows the Auth0 Settings page for a Single Page Application named "AutoStop". The "Settings" tab is selected. The page displays the following configuration details:

- Name:** AutoStop
- Domain:** srinix.auth0.com
- Client ID:** XvVLuuMQr3kKAR3ECAmBZOiPPyVYehvU
- Client Secret:** (Redacted)

A note at the bottom states: "The Client Secret is not base64 encoded." There is also a "Reveal client secret" link. On the right side, there is a "Continue with this tutorial" button. The left sidebar lists various Auth0 features: Dashboard, Applications (selected), APIs, SSO Integrations, Connections, Users, Rules, Hooks, Multi-factor Auth, Hosted Pages, Emails, Logs, Anomaly Detection, and Extensions.

The following are some important things to keep in mind:

- We need to update the Allowed Callback URLs, Allowed Web Origins, and Allowed Origins (CORS).
- If we do update the details for Allowed Web Origins and Allowed Origins, we will get a cross-origin request (CORS) error.

We have adjusted the required settings in **Auth0**, so we are good to implement **Auth0** in our application now.

In order to implement **Auth0** in our application, we will need to install a few modules, namely `auth0-js`, `auth0-lock`, and `angular2-jwt`:

```
D:\book_2\book\chapter14\user-authentication-auth0>npm install auth0-js auth0-lock angular2-jwt
npm [WARN] bootstrap@4.2.1 requires a peer of popper.js@^1.14.6 but none is installed. You must install peer dependencies yourself.
npm [WARN] angular2-jwt@0.2.3 requires a peer of @angular/core@'2.0.0||^4.0.0 but none is installed. You must install peer dependencies yourself.
npm [WARN] angular2-jwt@0.2.3 requires a peer of @angular/http@'2.0.0||^4.0.0 but none is installed. You must install peer dependencies yourself.
npm [WARN] angular2-jwt@0.2.3 requires a peer of rxjs@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm [WARN] optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\fsevents):
npm [WARN] notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ auth0-lock@11.12.1
+ auth0-js@9.8.2
+ angular2-jwt@0.2.3
added 31 packages from 92 contributors, updated 1 package and audited 40410 packages in 29.898s
found 0 vulnerabilities
```

In the preceding screenshot, using the `npm install` command, we installed the required Auth0 modules. Now, it's time to generate the services and components for our application.

First, we will need to generate our service; let's call it `authService`. We need to run the following command to generate our service:

```
ng g service services/auth
```

Upon successful execution of the preceding command, we should see the following output:

```
D:\book_2\book\chapter14\user-authentication-auth0>cd authentication-with-auth0
D:\book_2\book\chapter14\user-authentication-auth0\authentication-with-auth0>ng g service services/auth
CREATE src/app/services/auth.service.spec.ts (323 bytes)
CREATE src/app/services/auth.service.ts (133 bytes)
```

We can verify and confirm that our service has been generated, along with the spec file (the file used to write our test specifications). Now that we have created our service, it's time to generate the components. We will run the following commands using the `ng` CLI in order to generate the required components:

```
ng g c login
ng g c profile
```

Upon successful execution of the preceding commands, we should see the following output:

```
D:\book_2\book\chapter14\user-authentication-auth0\authentication-with-auth0>ng g c login
CREATE src/app/login/login.component.html (24 bytes)
CREATE src/app/login/login.component.spec.ts (621 bytes)
CREATE src/app/login/login.component.ts (266 bytes)
CREATE src/app/login/login.component.scss (0 bytes)
UPDATE src/app/app.module.ts (471 bytes)

D:\book_2\book\chapter14\user-authentication-auth0\authentication-with-auth0>ng g c profile
CREATE src/app/profile/profile.component.html (26 bytes)
CREATE src/app/profile/profile.component.spec.ts (635 bytes)
CREATE src/app/profile/profile.component.ts (274 bytes)
CREATE src/app/profile/profile.component.scss (0 bytes)
UPDATE src/app/app.module.ts (557 bytes)
```

In the preceding screenshot, we can verify and confirm that our required components, namely `login` and `profile`, have been generated successfully. Now, we are good to go ahead with implementing the functionality for our components.

To make our application beautiful, let's install the `bootstrap` CSS framework as well:

```
npm i bootstrap
```

We will also need to install the `jquery` module:

```
npm i jquery
```

Upon successful execution of the preceding command, we should see the following output:

```
D:\book_2\book\chapter14\user-authentication-auth0\authentication-with-auth0>npm i bootstrap
npm [WARN] bootstrap@4.2.1 requires a peer of jquery@1.9.1 - 3 but none is installed. You must install peer dependencies yourself.
npm [WARN] bootstrap@4.2.1 requires a peer of popper.js@^1.14.6 but none is installed. You must install peer dependencies yourself.
npm [WARN] optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\fsevents):
npm [WARN] notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ bootstrap@4.2.1
added 1 package from 2 contributors and audited 40179 packages in 21.706s
found 0 vulnerabilities
```

Super cool. Now, it's time to add a few links in the Nav component:

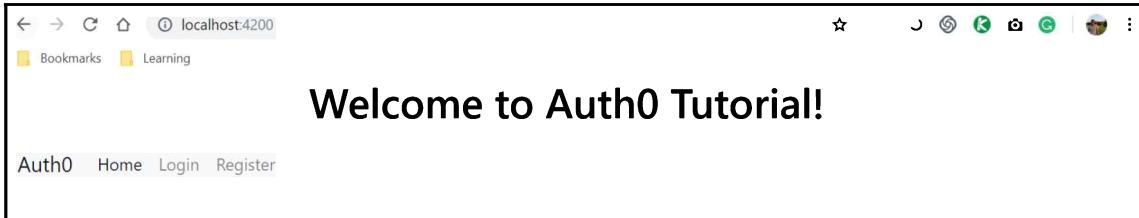
```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Auth0</a>
  <button class="navbar-toggler" type="button"
    data-toggle="collapse" data-target="#navbarSupportedContent"
    aria-controls="navbarSupportedContent" aria-expanded="false"
    aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home
          <span class="sr-only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" *ngIf="!authService.isLoggedIn();"
           (click)="authService.login()">Login</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" *ngIf="authService.isLoggedIn(); ">Profile</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" *ngIf="!authService.isLoggedIn(); "
           href="#">Register</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" *ngIf="authService.isLoggedIn()"
           (click)="authService.logout()">Logout</a>
      </li>
    </ul>
  </div>
</nav>
```

In the preceding code, the following important points should be noted:

- We are using the nav component of Bootstrap.
- We are adding a few links and attaching a click event, such as login and logout depending on the state of the user. If the user is logged in we will display logout link, else we will display register link
- We will implement these methods in our nav.component.ts file.
- We are using `*ngIf` to check whether the user is logged in and toggle the login and logout links accordingly

The output of the preceding code is shown in the following screenshot:



We will now need to work on the auth service that we have generated. In the `services/auth.service.ts` file, we need to import the required modules first and then add our methods, `login` and `logout`:

```
import { tokenNotExpired } from 'angular-jwt';
import { Auth0Lock } from 'auth0-lock';
```

Once we have imported the `Auth0Lock` and `TokenNotExpired` classes, we will create instances so that we can use them.

Take a look at the basic `Auth0Lock` object instance creation code:

```
var lock = new Auth0Lock(
  'YOUR_CLIENT_ID',
  'YOUR_AUTH0_DOMAIN'
);
```

In order to create a new object of the `Lock` class, we will need to pass the client ID and domain name to the instance.

Let's implement this in our `auth.service.ts` file:

```
public _idToken: string;
private _accessToken: string;
private _expiresAt: number;

lock = new
Auth0Lock('XvVLuuMOr3kKAR3ECAmBZOiPPyVYehvU', 'srinix.auth0.com', {
  allowedConnections: ["Username-Password-Authentication", "google-oauth2"],
  rememberLastLogin: false,
  socialButtonStyle: "big",
  languageDictionary: {"title": "Auth0"},
  language: "en",
  responseType: 'token id_token',
  theme: {}
});
```

In the preceding code, the following important points should be noted:

- We are creating three variables, namely `_idToken`, `_accessToken`, and `_expiresAt`.
- We are creating an instance of `Auth0Lock` and we need to pass params to the object.
- The `Auth0Lock` object will require two mandatory params to be passed. The first param is `ClientId`, and the second is the domain name.
- The third param includes options such as `allowedConnections`, `theme`, and so on, as it says they are optional.
- **Client Id and Domain** can be obtained from the **Auth0** app settings, as shown in the following screenshot:

The screenshot shows the Auth0 application settings interface for a single-page application named 'AutoStop'. The left sidebar lists various settings sections: Dashboard, Applications (selected), APIs, SSO Integrations, Connections, Users, Rules, Hooks, Multi-factor Auth, Hosted Pages, Emails, Logs, Anomaly Detection, and Extensions. The main content area displays the configuration for the 'AutoStop' application. It includes fields for Name (set to 'AutoStop'), Domain (set to 'srinix.auth0.com'), Client ID (set to 'XvVLuuMQr3kKAR3ECAmBZOiPPyVYehvU'), and Client Secret (a masked string). A note states: 'The Client Secret is not base64 encoded.' On the right side, there is a 'BILLING' button and a 'Continue with this tutorial' link.

We can now listen to events attached to the `lock` object:

```
constructor(private router: Router) {  
  
  this.lock.on('authenticated', (authResult: any) => {  
    localStorage.setItem("userToken", authResult.accessToken);  
    this.router.navigate(['/profile']);  
  });  
}
```

```
this.lock.on('authorization_error', error => {
  console.log('something went wrong', error);
});

}
```

In the preceding code, we are performing the following steps:

1. In the constructor method, we are listening to the on event for the authenticated and authorization_error states.
2. When we get an authenticated message from the lock instance, we are storing a localStorage item called userToken and setting accessToken as its value.
3. We are also listening to the error message and logging the message in the console.

Now, it's time to implement the login and logout methods:

```
login() {
  this.lock.show(function(err, profile, token) {
    console.log(err);
    console.log(profile);
    console.log(token);
  });
}
```

In the login method, we are calling the show method of the lock object. This will bring you to the dialog box of **Auth0**, with options to **Log In**, **Sign Up**, or **Don't remember your password?** The login dialog box will have social options if you selected any.

For the logout method, we just clear the userToken that we set when the user logs in and redirect the user back to the home login page:

```
logout(){
  localStorage.setItem('userToken', '');
  this.router.navigate(['/']);
}
```

Once we clear userToken, the application will know that the user is not logged in.

We have implemented the login and logout methods, but we also need a method to check whether the user is logged in or not:

```
isLoggedIn() {
  var token = localStorage.getItem('userToken');
  if(token != '')
  {
```

```
        return true;
    }
    else {
        return false;
    }
}
```

In the `isLoggedIn` method, we are checking whether the value of the `userToken` variable in local storage is set or not. If the value is set, it means that the user is logged in; otherwise, the user is not logged in.

Just import the service into our `app.component.ts` file and inject it into the constructor:

```
import { Component } from '@angular/core';
import { AuthService } from './services/auth.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'Auth0 Tutorial';
  userToken:string;

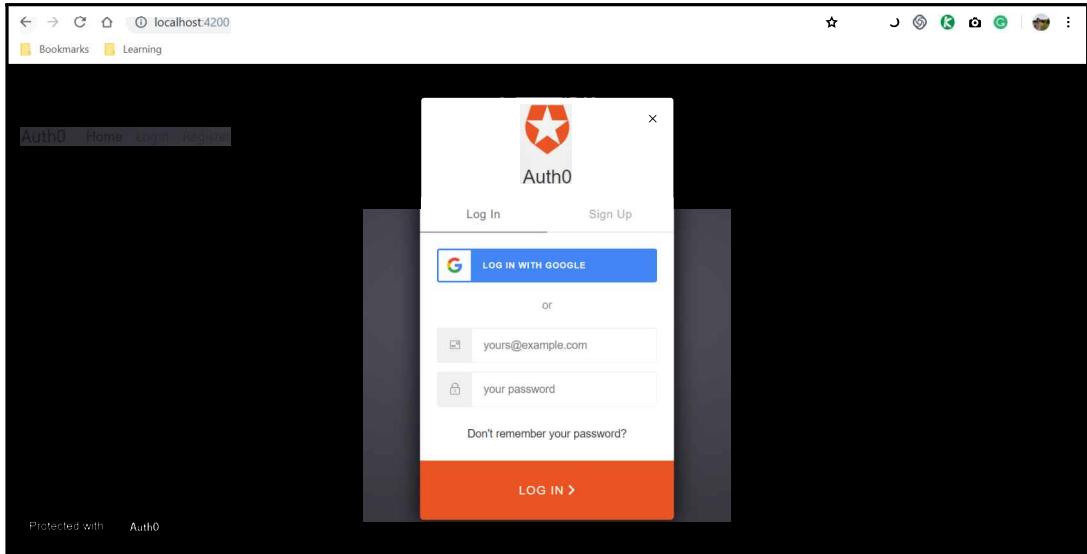
  constructor(private authService: AuthService) {}
}
```

That's it. Wasn't that simple?

We should see the following output:



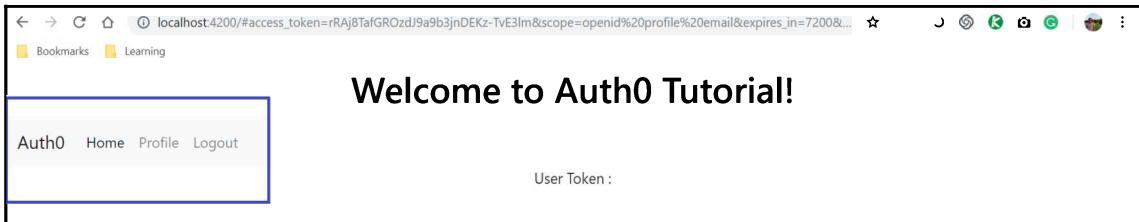
If we click on the **Login** link, we should see the **Auth0** dialog window pop up:



Now, go ahead and click on the **Sign Up** tab to create an account and, once registered successfully, you should see that the user has been added to the **Auth0** dashboard as well:

A screenshot of the Auth0 management interface at https://manage.auth0.com/#/users. The left sidebar shows navigation options like 'Dashboard', 'Applications', 'APIs', 'SSO Integrations', 'Connections', 'Users' (which is selected and highlighted in blue), 'Rules', 'Hooks', 'Multi-factor Auth', 'Hosted Pages', 'Emails', 'Logs', 'Anomaly Detection', 'Extensions', and 'Get support'. The main content area is titled 'Users' and contains a sub-header: 'An easy to use UI to help administrators manage user identities including password resets, creating and provisioning, blocking and deleting users.' Below this is a search bar with 'Search for users' and dropdown filters for 'Search by User' and 'Reset'. A table lists two users: 'test' (Email: test@localhost.com, Latest Login: 4 minutes ago, # of Logins: 27, Connection: Username-Pass...) and 'test123' (Email: test123@localhost.com, Latest Login: never, # of Logins: 0, Connection: Username-Pass...). A red '+ CREATE USER' button is located in the top right corner of the users table area. At the bottom right of the main content area is a 'Continue with this tutorial' button.

Once we log in successfully, we should see only the **Logout** link, as shown in the following screenshot:



When we click on the **Logout** link, the user should be taken back to the default landing page and should see the login and register options. Also, notice the params provided in the URL, such as `access_token` `expires_in`, and so on.

Awesome! We just implemented the entire user authentication using **Auth0** in our application.

Summary

In this chapter, we learned about some advanced Angular topics, from creating custom directives that are way too cool, to extending the behavior of our native HTML elements. We also created custom form validations, which are really useful when developing a really complex application with a lot of validations and compliance requirements. We dove into Angular single-page applications and looked at how they work and behave. We implemented user authentication in our Angular applications through native code.

We also learned how to build and implement a secure user authentication management system using the available frameworks, that is, Firebase and Auth0. We then learned to implement login, registration, and logout features to make sure we can secure the data and features of applications. Now that we have mastered the preceding concepts, we are good to implement a complete, wired end-to-end Angular application.

Now that we have learned how to develop our Angular applications, the only thing between our applications and real users is the deployment of our applications. That's the focus of our next chapter. In the next and final chapter of this book, we will learn all about deploying our Angular applications.

15

Deploying Angular Applications

Once you have completed building your application, it then has to be deployed to your test environment for the testing team to test it before deploying the application to your production environment for your users to use it. Although you can host your application virtually anywhere you like, there are three main ways in which you can package and deploy your Angular application. We'll explore these methods in this chapter:

- Deploying Angular applications
- Deploying composite Angular applications
- Deploying to GitHub Pages

Deploying Angular applications

Deploying our app is just as important as building the app itself. After all, our users need to access it; otherwise, it's not worth building, right?

Before we learn and explore how to deploy apps in detail, it's a prerequisite to have a server up and running. The server can be hosted on any operating system, be it Windows or Linux, and can be run on any application server, such as Apache Tomcat or IIS.

Alternatively, we can opt to choose any reliable cloud provider, such as AWS, Azure, or Bluehost, which offer hosting capabilities.

Technology stacks can vary from project to project; some clients prefer Java-based microservices, some may prefer .NET, and others may prefer Ruby on Rails. We will need to integrate our Angular applications with the backend APIs. The client-side code will mostly be Angular, which essentially means that Angular apps can be deployed and run on any server with any backend API services.

In this chapter, we are going to use the XAMPP server. XAMPP is a free distribution of Apache, MySQL, which makes it easy to set up our local server instantly and easily. You can download it at <https://www.apachefriends.org/download.html>.

Compilation options for Angular applications

I am sure by now you are aware that all the code we write for Angular is in TypeScript and that we will need to compile and generate deployable files using the `ng` command: `ng build`. This command will generate the corresponding equivalent JavaScript code that can just be copied into the environment we are trying to deploy.

Deploying Angular applications is very simple and easy. In real-time scenarios, the build and deploy commands are integrated into the build pipelines. A common practice is to have a single Angular project running in one repository. However, we can also run multiple projects in a single repository.

In this section, we will first learn about various compilation options we can consider for the deployment of our Angular applications. In the sections to follow, we will learn how to deploy a standalone application and also how to deploy composite Angular applications. Before we learn how to deploy our app, it's important to understand what happens when we build the application source code.

Angular has two compilation options, which are applied based on the commands and meta flags we use:

- Just-in-time compilation
- Ahead-of-time compilation

What is just-in-time compilation?

The Angular **just-in-time (JIT)** compilation refers to compiling the code in the browser at runtime. This is the default behavior whenever we run the `ng build` command:

```
ng build
```

This mechanism will add overhead to the request and Bootstrap time. The changes are reflected during runtime in our browser, which is great when developing an application. This option allows developers to quickly test changes while developing.

What is ahead-of-time compilation?

Angular's **ahead-of-time** (AOT) compilation means compiling the source TypeScript code, components, Angular HTML, libraries, and modules into the native JavaScript so that it can run on any browser smoothly. In other words, Angular will convert the code *before* it's downloaded by the browser.

Let's take a look at some of the benefits of AOT:

- Better security
- Faster rendering
- Smaller framework and application size
- Finds errors well in advance

Ahead Of Time or just AOT compilation is applied by default when we run the `ng build --prod` meta flag:

```
ng build --prod
```

Now that we have understood the different types of compilations offered by Angular, it's finally time to actually deploy an Angular app. In the next section, we will learn how to deploy the Angular application.

Deploying a standalone Angular application

Armed with knowledge about deployment and compilation strategies, it's time to deploy our Angular application. When we run the `ng build` and `ng build --prod` commands, the native JavaScript files are generated, which we can deploy to our server. This is good if we are trying to deploy a single project application.

In this section, we will learn how to deploy more complex use cases, such as when we have multiple projects in our Angular application.

We are going to keep our applications simple in order for our readers to be able to follow along with these steps easily. However, you can practice the deployment commands by deploying the Angular projects you have developed so far. Let's get started by creating a new Angular app:

1. To install Angular CLI, let's quickly use the following command:

```
npm i -g @angular/cli
```

The following screenshot shows the output of the preceding run command. We have just installed the Angular CLI, which we will use to generate our application:



A screenshot of a Microsoft Windows desktop showing a Visual Studio Code window titled "build-ng-app - Visual Studio Code". The window has a dark theme. In the center is a terminal window with the following text:

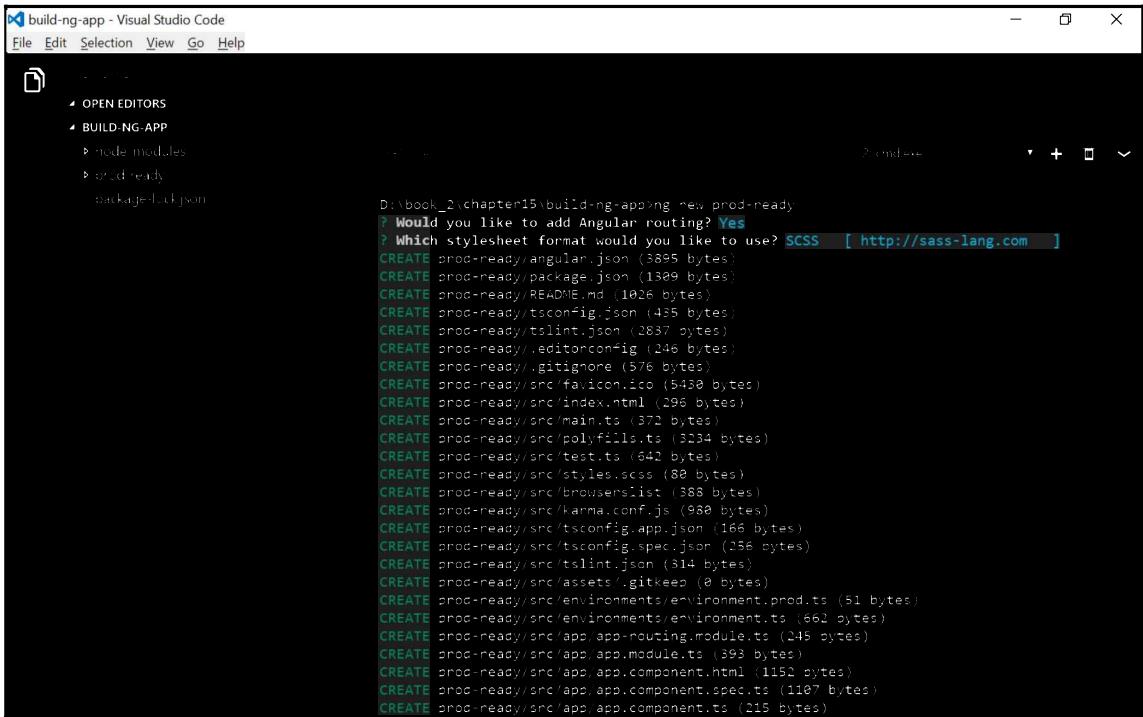
```
File Edit Selection View Go Help
- D:
OPEN EDITORS
BUILD-NG-APP
node_modules
package-lock.json
(c) 2018 Microsoft Corporation. All rights reserved.

D:\book_2\chapter15\build-ng-app>npm i -g @angular/cli@latest
C:\Users\bolttree\AppData\Roaming\npm>> C:\Users\bolttree\AppData\Roaming\npm\node_modules\@angular\cli\bin\ng
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.1.4 (node_modules\@angular\cli\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.1.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ @angular/cli@7.1.0
updated 1 package in 10.43s
```

2. Now that we have successfully installed the Angular CLI, it's time to create an Angular app and call it `prod-ready`:

```
ng new prod-ready
```

Using the preceding command, we have generated a new project. The following screenshot shows the output that is generated:



The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar has sections for 'OPEN EDITORS' and 'BUILD-NG-APP'. Under 'BUILD-NG-APP', there are sub-folders for 'node modules' and 'prod-ready'. A file named 'package-lock.json' is also visible. The main editor area displays a terminal window with the command 'D:\book_2\chapter15\build-ng-app>ng new prod-ready'. It asks 'Would you like to add Angular routing?' with options 'Yes' and 'No'. The user selects 'Yes'. The terminal then lists the files being created, including 'prod-ready/angular.json', 'prod-ready/package.json', 'prod-ready/README.md', 'prod-ready/tsconfig.json', 'prod-ready/tslint.json', 'prod-ready/editorconfig', 'prod-ready/.gitignore', 'prod-ready/src/favicon.ico', 'prod-ready/src/index.html', 'prod-ready/src/main.ts', 'prod-ready/src/polyfills.ts', 'prod-ready/src/test.ts', 'prod-ready/src/styles.scss', 'prod-ready/src/karma.conf.js', 'prod-ready/src/tsconfig.app.json', 'prod-ready/src/tsconfig.spec.json', 'prod-ready/src/tslint.json', 'prod-ready/src/assets/.gitkeep', 'prod-ready/src/environments/environment.prod.ts', 'prod-ready/src/environments/environment.ts', 'prod-ready/src/app/app-routing.module.ts', 'prod-ready/src/app/app.module.ts', 'prod-ready/src/app/app.component.html', 'prod-ready/src/app/app.component.spec.ts', and 'prod-ready/src/app/app.component.ts'. The total size of the files is approximately 1.026 MB.

Beautiful! We have our newly generated application.

3. Now, let's navigate to the `prod-ready` application folder, as follows:

```
cd prod-ready
```

4. All done. We are not going to change or add any new components. For now, I want you to understand the simplest way to deploy an app. Now, fire up the app using the `ng serve` command:

```
ng serve
```

The preceding command will get the application started, and we should see the output displayed in the following screenshot:

A screenshot of the Visual Studio Code interface. On the left, the file tree shows a project structure with files like angular.json, tsconfig.json, and tsconfig.app.json. The right side features a terminal window with the following text:

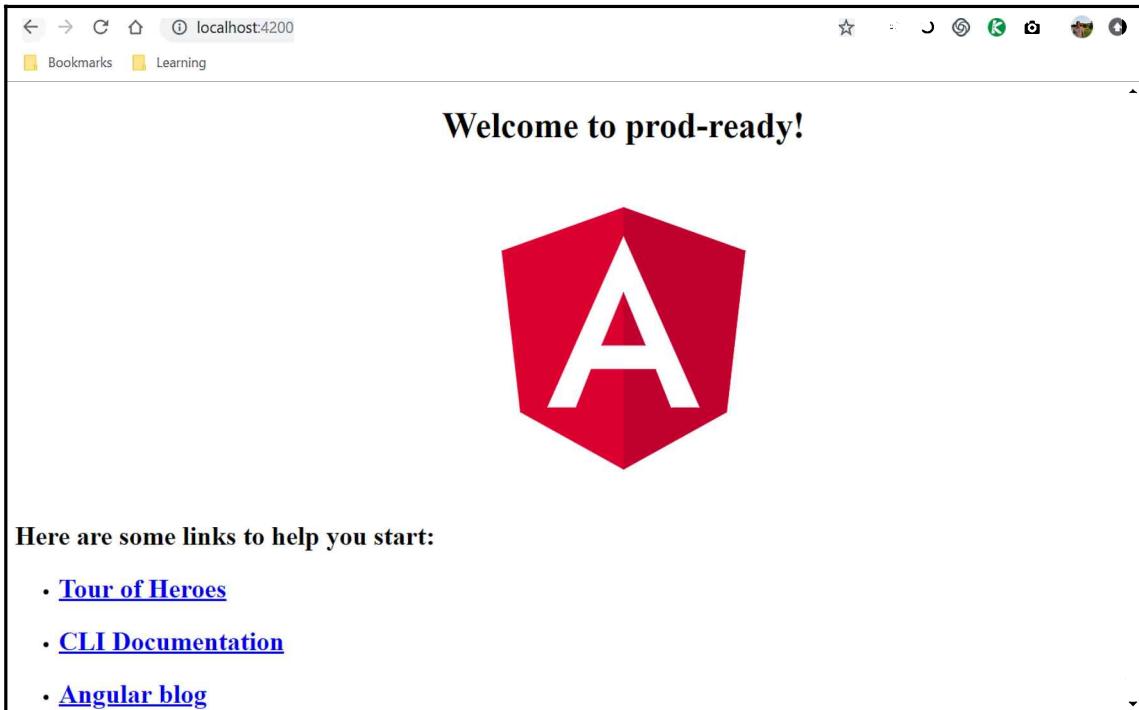
```
tsconfig.json - build-ng-app - Visual Studio Code
File Edit Selection View Go Help
OPEN EDITORS
angular.json
tsconfig.json
BUILD-NG-APP
node modules
prod ready
src
  editorconfig
  gitignore
  angular.json
  package.json
  package-lock.json
  README.md
  tsconfig.json
  tsconfig.app.json
  package-lock.json

tsconfig.json
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/tslint.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in tsconfig.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in tslint.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in tsconfig.app.json.
Successfully initialized git.

D: book_2 chapter15/build-ng-app$ prod-ready
D: book_2 chapter15/build-ng-app$ prod-ready ng serve
*** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200 ***

Date: 2018-11-28T19:16:04.304Z
Hash: 58c7d8bd4627be2034b0
Time: 15828ms
chunk {main} main.js, main.js.map (main) 11.5 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 223 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.08 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 16.7 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.67 MB [initial] [rendered]
i Twdm: Compiled successfully.
```

5. Launch the browser and then type `http://localhost:4200`. The default vanilla application should be displayed as follows:



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Awesome. So far, so good. We got our app working on our local environment and now it's time to deploy it to our application—that's right!

To make you comfortable with the whole deployment process, we will deploy the vanilla application as it is, without making any changes.

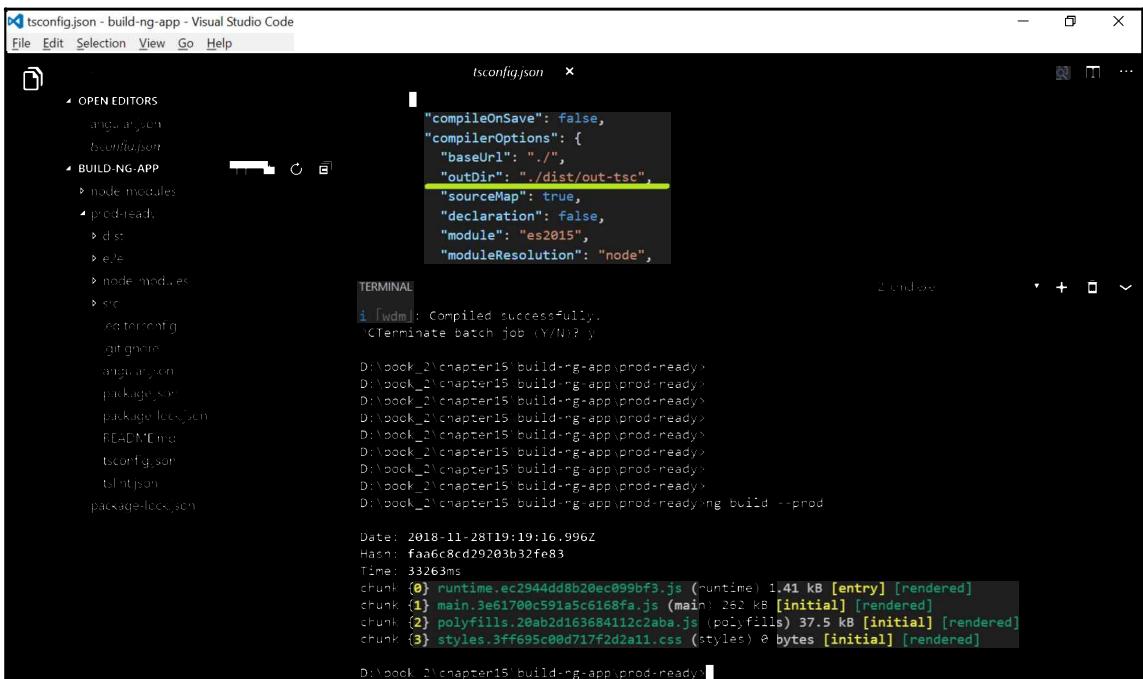
6. To deploy, run the following `ng` command:

```
ng build --prod
```

Once the command runs successfully, you should see that the following folders and files have been created. Let's take a look at some of the important things to note:

- You should notice a new folder called `dist/<defaultProject>`.
- You should also notice the following files created inside the `dist` folder:
 - `runtime`
 - `main`
 - `polyfills`
 - `styles`

The output of the preceding `build` command is given as follows. The output will be under the `dist` folder with the same application name:



The screenshot shows a Visual Studio Code interface. On the left, the file tree (OPEN EDITORS) shows files like `angular.json`, `tsconfig.json`, and `tsconfig.app.json`. In the center, the terminal window displays the command `ng build` running. The output shows the build process, including the creation of several files in the `dist` directory. The files listed are `runtime.js`, `main.js`, `polyfills.js`, and `styles.css`. The terminal also shows the date and time of the build (2018-11-28T19:19:16.996Z), the hash of the build (faa6c8cd29203b32fe83), and the total time taken (33263ms).

```
tsconfig.json - build-ng-app - Visual Studio Code
File Edit Selection View Go Help
tsconfig.json x
OPEN EDITORS
  angular.json
  tsconfig.json
  tsconfig.app.json
BUILD-NG-APP
  node_modules
  prod-readies
  dist
  e2e
  node_modules.es
  src
  tsconfig
  gitignore
  angular.json
  package.json
  package-lock.json
  README.md
  tsconfig.json
  tsint.json
  package-lock.json

tsconfig.json
"compileOnSave": false,
"compilerOptions": {
  "baseUrl": "./",
  "outDir": "./dist/out-tsc",
  "sourceMap": true,
  "declaration": false,
  "module": "es2015",
  "moduleResolution": "node",
  "emitDecoratorMetadata": true,
  "experimentalDecorators": true,
  "target": "es5",
  "typeRootPath": "./node_modules/@types"
}

TERMINAL
i [wdm]: Compiled successfully.
?C[Terminate batch job (Y/N)?] y
D:\book_2\chapter15\build-ng-app.prod-ready>
Date: 2018-11-28T19:19:16.996Z
Hash: faa6c8cd29203b32fe83
Time: 33263ms
chunk {0} runtime.ec2944dd8b20ec099bf3.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.3e61700c591a5c6168fa.js (main) 36 kB [initial] [rendered]
chunk {2} polyfills.20ab2d163684112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
D:\book_2\chapter15\build-ng-app.prod-ready
```

7. We don't have to necessarily use the default folder name; that is, we can provide the output path and folder name as an argument and Angular will generate the code in that folder. It's easy to customize the output directory where we would want our files to be generated:

```
ng build --prod --output-path dist/compiled
```

Run the preceding command and we should see our custom folder and files generated in our folder. In the preceding command, we specified that we want our files to be generated in the folder named `compiled`, and we provided the path. The following is a screenshot after the command has run successfully:

The screenshot shows a Visual Studio Code interface. On the left, the file tree displays a project structure under 'BUILD-NG-APP'. A yellow box highlights the 'dist/compiled' folder, which contains files like '3rdpartylicenses.txt', 'favicon.ico', 'index.html', 'main.3e61700c591a5c6168fa.js', 'polyfills.20ab2d163684112c2aba.js', 'runtime.ec2944dd8b20ec099bf3.js', and 'styles.3ff695c00d717f2d2a11.css'. To the right of the file tree is the 'angular.json' file content, and below it is a terminal window showing the build command and its output.

```

{
  "projectType": "application",
  "prefix": "app",
  "schematics": {
    "@schematics/angular:component": {
      "styleext": "scss"
    }
  },
  "architect": {
    "build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
        "outputPath": "dist/prod-ready",
        "index": "src/index.html",
        "main": "src/main.ts",
        "tsConfig": "tsconfig.app.json"
      },
      "configurations": {
        "prod": {
          "fileReplacements": [
            {
              "replace": "src/environments/environment.ts",
              "with": "src/environments/environment.prod.ts"
            }
          ],
          "optimization": true,
          "outputHashing": "all",
          "sourceMap": false,
          "extractCss": true,
          "namedChunks": true,
          "aot": true,
          "extractLicenses": true,
          "treeshake": true
        }
      }
    }
  }
}

TERMINAL
D:\book_2\chapter15\build-ng-app>ng build --prod --output-path dist/compiled
chunk {0} runtime.ec2944dd8b20ec099bf3.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.3e61700c591a5c6168fa.js (main) 262 kB [initial] [rendered]
chunk {2} polyfills.20ab2d163684112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
Date: 2018-11-28T19:29:14.078Z
Hash: faabc8cd29203b32fe83
Time: 3555ms
chunk {0} runtime.ec2944dd8b20ec099bf3.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.3e61700c591a5c6168fa.js (main) 262 kB [initial] [rendered]
chunk {2} polyfills.20ab2d163684112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
D:\book_2\chapter15\build-ng-app>

```

That's all we need to do to generate and deploy our Angular application. Just copy all the files to the root directory on your server and that's it.

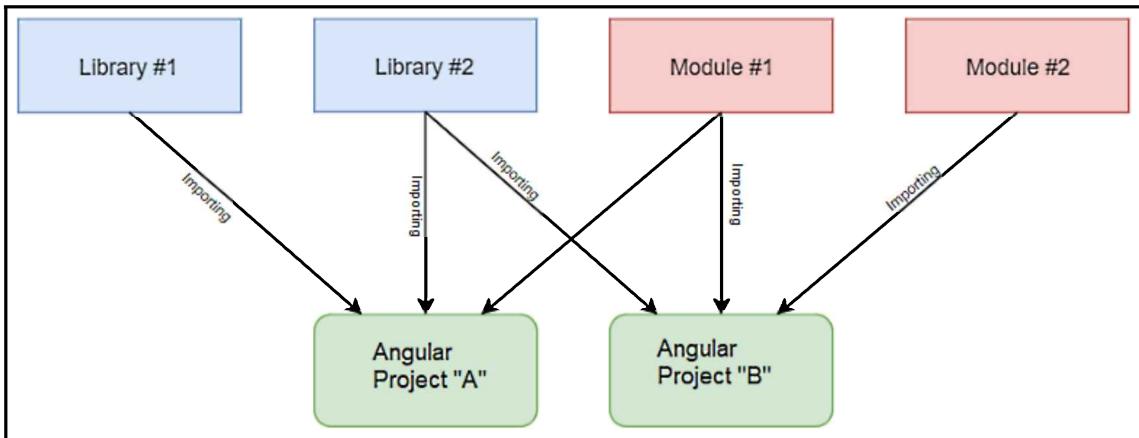
In the next section, we will learn how to deploy a more complex architecture of an Angular application, and we will then go on to deploy the composite application in multiple ways.

Deploying composite Angular applications

In the previous section, we learned how to deploy a standalone Angular application, which is fairly straightforward. However, we can be in situations where we may have to build and deploy multiple applications all running in a single repository. Is it possible? Certainly. In this section, we will create an Angular repository with multiple projects and learn how to deploy a composite application.

Creating and deploying multiple Angular applications

In a more realistic real-world application, we will need to run multiple Angular applications, which will be composed of multiple projects, libraries, modules, and microservices, as shown in the following diagram:



In the preceding diagram, some of the important things to note are detailed as follows:

- There are multiple Angular projects and applications.
- **Library #1** and **Library #2** can be reused in multiple projects simply by importing the libraries.
- During the development phase, we will create multiple modules that can also be reused in multiple projects.

So, let's jump right into it and create multiple projects, libraries, and modules. Finally, we'll package the app in different ways. So, let's begin by getting our Angular application up and running:

1. First things first. We will need to generate an application, and we are going to use Angular CLI to generate the application. We will first need to install the Angular CLI using the following command:

```
npm install @angular/cli
```

Upon successful execution of the preceding command, we should see the following output:

```
D:\book_2\testing>npm install @angular/cli
npm WARN saveError ENOENT: no such file or directory, open 'D:\book_2\testing\package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open 'D:\book_2\testing\package.json'
npm WARN testing No description
npm WARN testing No repository field.
npm WARN testing No README data
npm WARN testing No license field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ @angular/cli@7.1.2
added 292 packages from 178 contributors and audited 16182 packages in 35.674s
found 0 vulnerabilities
```

2. Now that we have installed Angular CLI, let's create the app using the following command. We are calling it shopping-cart. Now, run the following ng command to generate the new project:

```
ng new shopping-cart
```

Using the preceding command, we are generating a new application called shopping-cart. The output of the preceding command is given as follows:

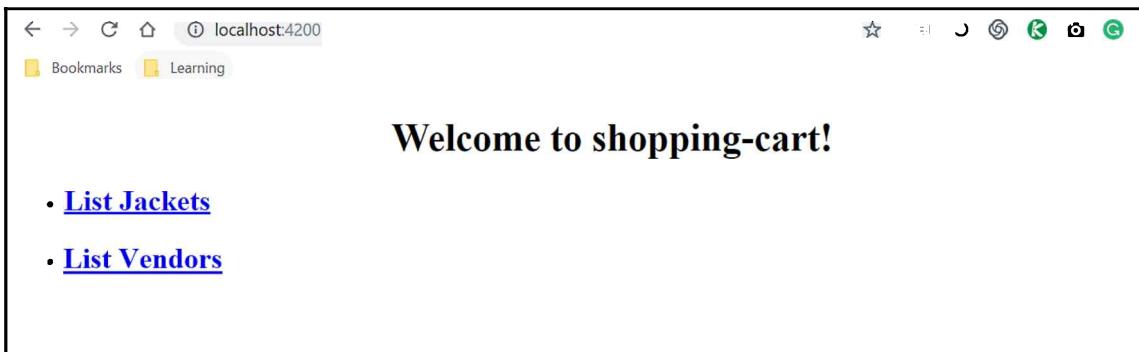
```
D:\book_2\testing>ng new shopping-cart
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ http://sass-lang.com ]
```

CREATE shopping-cart/angular.json (3922 bytes)
CREATE shopping-cart/package.json (1312 bytes)
CREATE shopping-cart/README.md (1029 bytes)
CREATE shopping-cart/tsconfig.json (435 bytes)
CREATE shopping-cart/tslint.json (2824 bytes)
CREATE shopping-cart/.editorconfig (246 bytes)
CREATE shopping-cart/.gitignore (576 bytes)
CREATE shopping-cart/src/favicon.ico (5430 bytes)
CREATE shopping-cart/src/index.html (299 bytes)
CREATE shopping-cart/src/main.ts (372 bytes)
CREATE shopping-cart/src/polyfills.ts (3234 bytes)
CREATE shopping-cart/src/test.ts (642 bytes)
CREATE shopping-cart/src/styles.scss (80 bytes)
CREATE shopping-cart/src/browserlist (388 bytes)
CREATE shopping-cart/src/karma.conf.js (980 bytes)
CREATE shopping-cart/src/tsconfig.app.json (166 bytes)
CREATE shopping-cart/src/tsconfig.spec.json (256 bytes)
CREATE shopping-cart/src/tslint.json (314 bytes)
CREATE shopping-cart/src/assets/.gitkeep (0 bytes)
CREATE shopping-cart/src/environments/environment.prod.ts (51 bytes)
CREATE shopping-cart/src/environments/environment.ts (662 bytes)
CREATE shopping-cart/src/app/app-routing.module.ts (245 bytes)
CREATE shopping-cart/src/app/app.module.ts (393 bytes)

3. We have now created our new app called shopping cart. Let's modify `app.component.html` and add two `routerLink` hyperlinks named `list-jackets` and `list-vendors`:

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
<ul>
  <li>
    <h2><a routerLink="/list-jackets" class="nav-link">List
      Jackets</a></h2>
  </li>
  <li>
    <h2><a routerLink="/list-vendors" class="nav-link">List
      Vendors</a></h2>
  </li>
</ul><router-outlet></router-outlet>
```

In the preceding code, we have created two links in the `app.component.html` file. The result is displayed as follows:



So far, so good. Essentially, we have an Angular application up and running. Now, we are going to learn how to run and deploy multiple Angular projects inside the same repository. In order to do so, we will follow these steps:

1. Let's create a new application in the same repository using the following command. We are generating a new application called `jackets`:

```
ng g application jackets
```

We are creating a new application using the `ng` command, which we'll name `jackets`. We should see the following output:

```
D:\book_2\testing\shopping-cart>ng g application jackets
CREATE projects/jackets/src/favicon.ico (5430 bytes)
CREATE projects/jackets/src/index.html (294 bytes)
CREATE projects/jackets/src/main.ts (372 bytes)
```

- Woohoo! With the Angular CLI schematics, it's really simple to create multiple projects inside the same app. Take a look at the files that have been autogenerated and some of the files that have been updated by the Angular CLI for us:

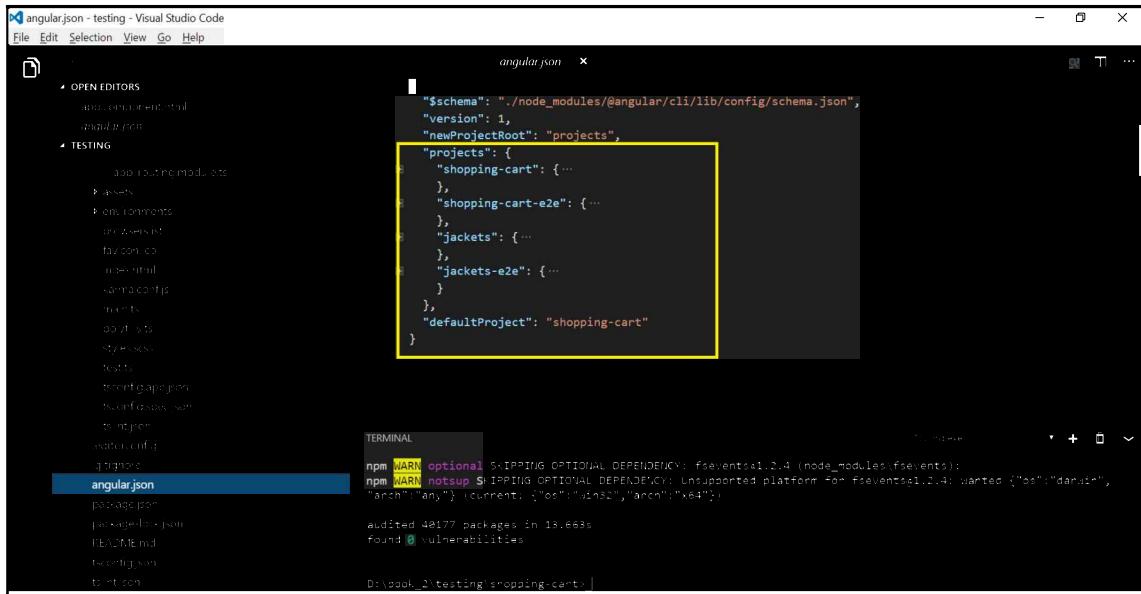
The screenshot shows a code editor interface with a sidebar navigation and two main panes: a preview pane and a terminal pane.

- File Structure (Left):**
 - OPEN EDITORS: app.component.html
 - TESTING: node_modules
 - shopping-cart: node_modules, packages
 - projects: jackets (selected), jackets-e2e
 - src: app, environments
 - assets: icon, images
 - environments: environment.ts
- Preview Pane (Top Right):** Displays the content of `app.component.html`, which includes a placeholder message: `<!--The content below is only a placeholder and can be replaced.-->`.
- Terminal Pane (Bottom Right):** Shows the command-line output of the `ng g application jackets` command, which creates three files in the `jackets` directory:
 - `CREATE projects/jackets/src/favicon.ico (5430 bytes)`
 - `CREATE projects/jackets/src/index.html (294 bytes)`
 - `CREATE projects/jackets/src/main.ts (372 bytes)`

If you look carefully, you'll notice that the following are some of the important things that have happened to our application structure and files:

- A new folder called `Projects` is auto-created and its corresponding entry is generated in the `angular.json` file.
- In the `Projects` folder, we will see the new `Jackets` project with the same default vanilla application files that have been generated.

3. Now, to verify whether the new `Jackets` project was added, let's check out the `Angular.json` file:



```
angular.json
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "shopping-cart": {...},
    "shopping-cart-e2e": {...},
    "jackets": {...},
    "jackets-e2e": {...}
  },
  "defaultProject": "shopping-cart"
}
```

You will notice inside the `Angular.json` file that we have project-specific entries for `shopping-cart`, `shopping-cart-e2e`, `jackets`, and `jackets-e2e`. Beautiful. Technically speaking, we are now running two apps inside the same repository.

4. It's now time to extend our application by adding a few components, libraries, and modules. First, we will need to create a component inside our `jackets` project. Run the following `ng` command to generate the component:

```
ng g c jacket-list --skip-import
```

Run the preceding command and we should see the component and respective files generated. We should see the following output:

```
jacket-list.component.ts - testing - Visual Studio Code
File Edit Selection View Go Help
jacket-list.component.ts x
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-jacket-list',
  templateUrl: './jacket-list.component.html',
  styleUrls: ['./jacket-list.component.scss']
})
export class JacketListComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}

TERMINAL
D:\book_2\testing\shopping-cart\projects>ng generate component jacket-list --skip-import
CREATE projects/jacket-list/jacket-list.component.html (30 bytes)
CREATE projects/jacket-list/jacket-list.component.spec.ts (657 bytes)
CREATE projects/jacket-list/jacket-list.component.ts (289 bytes)
CREATE projects/jacket-list/jacket-list.component.scss (0 bytes)

D:\book_2\testing\shopping-cart\projects>
```

- Now that we have created a new component inside the `Jackets` project, it's time to add it to `app-routing.module.ts` so that it's available to use across the `Jackets` project.

In the following code snippet, we are importing the newly created component inside the `app-routing.module.ts` file:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AppComponent } from './app.component';
import { JacketListComponent } from
'../../projects/jackets/src/app/jacket-list/jacket-list.component';
import { VendorsComponent } from
'../../projects/vendors/src/lib/vendors.component';
```

- After importing the component, it's time to create a route for our component:

```
const routes: Routes = [
  {
    path:'home',
    component:AppComponent
  },
  {
    path:'list-jackets',
    component:JacketListComponent
  },
```

```
{  
  path:'list-jackets',  
  component:JacketListComponent  
}  
];
```

In the preceding code snippet, we are creating `list-jackets` and `list-vendors` routes that are mapped to the respective `JacketListComponent` and `VendorsComponent` components. Here are two important things to note in the preceding code snippet:

- We are running multiple Angular projects.
 - We are linking components from various projects inside each other.
7. We have already added the router links to `app.component.html`. Now, let's fire up our application by running the `ng serve` command:
- ```
ng serve
```
8. Launch the `http://localhost:4200` browser and we should see the following output displayed:



So, now we have two apps running and we have components that are shared across different projects.

Great. Now, why don't we add a few libraries that we can share between multiple projects? Let's begin:

1. We will create a new Angular library called vendors. We will use the ng command and call the library vendors. Let's run the following command to generate the library:

```
ng g library vendors --prefix=my
```

On running the preceding command successfully, we should see the following output:

```
D:\book_2\testing\shopping-cart\projects>ng g library vendors --prefix=my
CREATE projects/vendors/karma.conf.js (968 bytes)
CREATE projects/vendors/ng-package.json (154 bytes)
CREATE projects/vendors/package.json (137 bytes)
CREATE projects/vendors/tsconfig.lib.json (726 bytes)
CREATE projects/vendors/tsconfig.spec.json (246 bytes)
CREATE projects/vendors/tslint.json (245 bytes)
CREATE projects/vendors/src/test.ts (700 bytes)
CREATE projects/vendors/src/projects.ts (159 bytes)
CREATE projects/vendors/src/lib/vendors.module.ts (229 bytes)
CREATE projects/vendors/src/lib/vendors.component.spec.ts (635 bytes)
CREATE projects/vendors/src/lib/vendors.component.ts (258 bytes)
CREATE projects/vendors/src/lib/vendors.service.spec.ts (338 bytes)
CREATE projects/vendors/src/lib/vendors.service.ts (136 bytes)
```

- Once the library is generated, Angular CLI will create the following folders and files:

The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** On the left, it shows the project structure. A yellow box highlights the `src` folder under `app`, which contains files like `karma.conf.js`, `ng-package.json`, `package.json`, `tsconfig.lib.json`, `tsconfig.spec.json`, and `tslint.json`.
- Editor:** The main editor area displays the `app.component.html` file content:

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
 <h1>
 Welcome to {{ title }}!
 </h1>
</div>

 <h2>List Jackets</h2>

 <h2>List Shoes</h2>


```
- Terminal:** At the bottom, the terminal window shows the command output:

```
to help us maintain this package.

https://opencollective.com/ng-pa...
```

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node\_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin", "arch":"any"} (current: {"os":"win32", "arch":"x64"})

added 121 packages from 59 contributors and audited 47678 packages in 31.04s
found 0 vulnerabilities

- Here are some important things to note once the command runs successfully:

- Under `Projects`, a new `Vendors` library project is created.
- Angular will also make the necessary changes and entries in the `Angular.json` file.
- Note that `projectType` is of the `library` type.

The following screenshot shows the data displayed for the newly created library project:

The screenshot shows the Visual Studio Code interface with the file `angular.json` open in the center editor pane. The left sidebar displays the project structure, including the `shopping-cart` application and the `jackets` library project. The `angular.json` file contains configuration for the library, specifically for the `vendors` section, which is highlighted with a yellow box. The code includes details about the root directory, source root, project type (library), prefix, and architect build options. The status bar at the bottom indicates the file is on branch `master`, has 0 changes, and is at line 270, column 19.

```
{
 "devServerTarget": "jackets:serve:production"
},
"lint": {
 "builder": "@angular-devkit/build-angular:tslint",
 "options": {
 "tsConfig": "projects/jackets-e2e/tsconfig.e2e.json",
 "exclude": [
 "**/node_modules/**"
]
 }
},
"vendors": {
 "root": "projects/vendors",
 "sourceRoot": "projects/vendors/src",
 "projectType": "library",
 "prefix": "my",
 "architect": {
 "build": [
 {
 "builder": "@angular-devkit/build-ng-packagr:build",
 "options": {
 "tsConfig": "projects/vendors/tsconfig.lib.json",
 "project": "projects/vendors/ng-package.json"
 }
 }
],
 "test": {
 "builder": "@angular-devkit/build-angular:karma",
 "options": {}
 }
 }
}
```

4. Now, open the `vendors` folder and, under `src/lib`, edit the `vendors.component.ts` file and add some fancy text:

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

@Component({
 selector: 'my-vendors',
 template: `
 <p>
 vendors works!
 </p>
 `,
 styles: []
})
export class VendorsComponent implements OnInit {
 constructor() { }
```

```
ngOnInit() {
}

}
```

5. Remember, we have created the router link for the vendor component earlier, so we should see the changes reflected in the application:



Now that we have built an Angular app that has multiple projects, libraries, and routing systems to share different components, it's time to deploy the app.

Deployment is simple and is just like what we did for a standalone app:

```
ng build --prod --base-href "http://localhost/deploy-angular-app/"
```

Once you run the command, here are some important things that will happen:

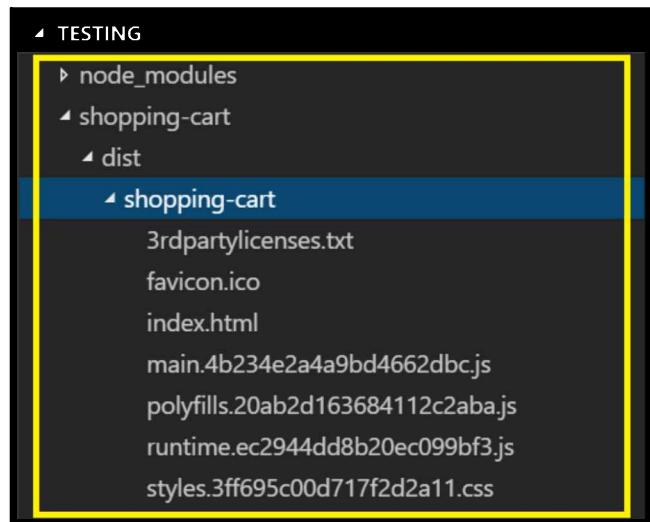
- To generate the final deployment files, we are running the `ng build` command.
- We are using the `--prod` meta flag, to which we will apply AOT compilation while compiling.
- Most importantly, we need to pass the `--base-href` meta flag, which will point to the server's root folder/path.



Without a proper `--base-href` value, Angular applications will not work properly and will give you errors to link the generated files.

```
D:\book_2\testing\shopping-cart\projects>ng build --prod --base-href "http://localhost/deploy-angular-app/"
Date: 2018-12-13T16:28:12.789Z
Hash: b002f537bcd0f4248c18
Time: 20398ms
chunk {0} runtime.ec2944dd8b20ec099bf3.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.4b234e2a4a9bd4662dbc.js (main) 265 kB [initial] [rendered]
chunk {2} polyfills.20ab2d163684112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
```

From the previous section, we already know that after we run the `build` command, Angular will generate the compiled folders and files, as shown in the following screenshot:

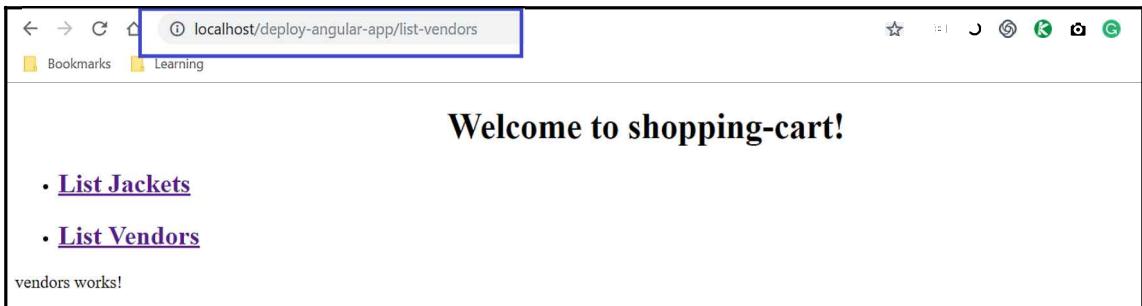


Here are some important points to note from the preceding screenshot:

- The command will generate the output of compiled files that have multiple projects, libraries, and components.
- Carefully consider the `--base-href` value we have set. We are running XAMPP locally, hence the path is pointing to the localhost.

Now, let's copy all the code from the `dist` folder and paste it into our XAMPP folder.

Launch the Angular application using the local server and you should see the output displayed as follows:



That's really cool! Even so, we can improve this a lot. In a more realistic setup, any large Angular implementations will have feature teams, and the library or module developed by one team should be easily shareable with other teams as a module. That's where writing reusable modules comes into the picture. We are going to learn how to distribute Angular modules as `npm` modules.

## Packing the Angular project as an `npm` package

Now, let's learn how to export our Angular project as an `npm` module. We will continue to use the same `vendors` library that we created in the previous example:

1. Note that we do *not* wish to deploy the entire application, rather, we only want to deploy the `vendors` library. We will use the same `ng build` command to build the `vendors` Angular project:

```
ng build vendors
```

2. Once the command is successfully executed, we will see that Angular will generate the compiled files for our `vendors` project under the `dist` folder, as follows:

The screenshot shows a Visual Studio Code interface. On the left is the 'File Explorer' sidebar with a tree view of files and folders. A yellow box highlights the 'dist' folder within the 'node\_modules/shopping-cart' entry. The main editor area shows the 'app.component.html' file with some basic Angular template code. To the right is the 'Terminal' pane, which displays the command 'ng build vendors' being run and its output, which includes building Angular packages, compiling TypeScript sources through ngc, and bundling to UMD.

```

app.component.html - testing - Visual Studio Code
File Edit Selection View Go Help
OPEN EDITORS
app.component.html shopping-cart\src\app
angular.json shopping-cart
app.component.html shopping-cart\projects\jackets\...
app-routing.module.ts shopping-cart\src\app
vendors.component.ts shopping-cart\projects\vendors\...
app.module.ts shopping-cart\src\app
app.components.ts shopping-cart\projects\jackets\src\...
TESTING
> node_modules
 > shopping-cart
 > dist
 > shopping-cart
 > vendors
 > e2e
 > node_modules
 > projects
 > jackets
 > jackets-e2e
 > vendors
 > src
 karma.conf.js
 ng-package.json
 package.json
 tsconfig.lib.json
 tsconfig.spec.json
TERMINAL
1: cmd.exe
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
D:\book_2\testing\shopping-cart\projects>ng build vendors
Building Angular Package
Building entry point 'vendors'
Compiling TypeScript sources through ngc
Bundling to FESM2015
Bundling to FESM5
Bundling to UMD
Minifying UMD bundle
Copying declaration files
Copying declaration files
Writing package metadata
Removing scripts section in package.json as it's considered a potential security vulnerability.
Built vendors
Built Angular Package
- from: D:\book_2\testing\shopping-cart\projects\vendors
- to: D:\book_2\testing\shopping-cart\dist\vendors
D:\book_2\testing\shopping-cart\projects>

```

3. Navigate to the `dist/vendors` folder and run the following command:

```
npm pack
```

We are using the `npm pack` command to generate a package out of the current folder, which is compiled of files from the `vendors` project. We should see the following output:

```
D:\book_2\testing\shopping-cart\dist\vendors>npm pack
npm notice
npm notice package: vendors@0.0.1
npm notice === Tarball Contents ===
npm notice 512B package.json
npm notice 121B projects.d.ts
npm notice 80B vendors.d.ts
npm notice 1.3kB vendors.metadata.json
npm notice 3.1kB bundles/vendors.umd.js
npm notice 1.3kB bundles/vendors.umd.js.map
npm notice 998B bundles/vendors.umd.min.js
npm notice 1.3kB bundles/vendors.umd.min.js.map
npm notice 1.4kB esm2015/lib/vendors.component.js
npm notice 1.4kB esm2015/lib/vendors.module.js
npm notice 1.3kB esm2015/lib/vendors.service.js
npm notice 921B esm2015/projects.js
```

- Upon successful execution, we will see the `vendors-0.0.1.tgz` file created in the folder. We can now distribute this file as an `npm` package, which can be reused across any projects:

| Name             | Date modified      | Type                   | Size |
|------------------|--------------------|------------------------|------|
| bundles          | 12/13/2018 2:48 PM | File folder            |      |
| esm5             | 12/13/2018 2:48 PM | File folder            |      |
| esm2015          | 12/13/2018 2:48 PM | File folder            |      |
| fesm5            | 12/13/2018 2:48 PM | File folder            |      |
| fesm2015         | 12/13/2018 2:48 PM | File folder            |      |
| lib              | 12/13/2018 2:48 PM | File folder            |      |
| package          | 12/13/2018 2:48 PM | JSON Source File       | 1 KB |
| projects.d       | 12/13/2018 2:48 PM | TypeScript Source File | 1 KB |
| vendors.d        | 12/13/2018 2:48 PM | TypeScript Source File | 1 KB |
| vendors.metadata | 12/13/2018 2:48 PM | JSON Source File       | 2 KB |
| vendors-0.0.1    | 12/13/2018 2:51 PM | tgz Archive            | 6 KB |

- Let's now give it a test drive, by installing the newly generated `npm` module into our application. To install the package, run the `npm install` command by pointing to `vendors-0.0.1.tgz`:

```
npm install dist\vendors\vendors-0.0.1.tgz
```

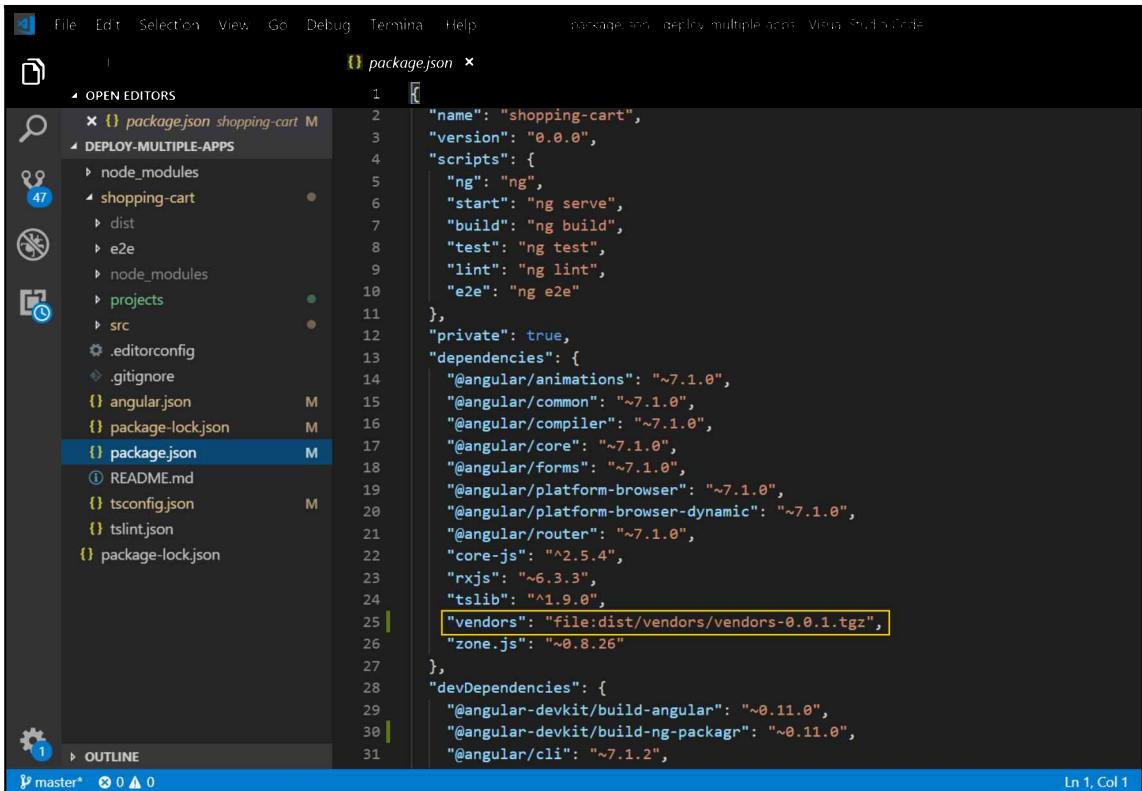
- Upon completion, we should see the following output informing us that the package has been added:

```
D:\book_2\testing\shopping-cart>
D:\book_2\testing\shopping-cart>npm install dist\vendors\vendors-0.0.1.tgz
npm WARN The package tslib is included as both a dev and production dependency.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ vendors@0.0.1
added 1 package and audited 47680 packages in 17.751s
found 0 vulnerabilities

D:\book_2\testing\shopping-cart>
```

7. We can also verify whether the package was added successfully to the package.json file. We should see an entry in package.json displayed as follows:



The screenshot shows the Visual Studio Code interface with the 'package.json' file open in the center editor. The file content is as follows:

```
1 "name": "shopping-cart",
2 "version": "0.0.0",
3 "scripts": {
4 "ng": "ng",
5 "start": "ng serve",
6 "build": "ng build",
7 "test": "ng test",
8 "lint": "ng lint",
9 "e2e": "ng e2e"
10 },
11 },
12 "private": true,
13 "dependencies": {
14 "@angular/animations": "~7.1.0",
15 "@angular/common": "~7.1.0",
16 "@angular/compiler": "~7.1.0",
17 "@angular/core": "~7.1.0",
18 "@angular/forms": "~7.1.0",
19 "@angular/platform-browser": "~7.1.0",
20 "@angular/platform-browser-dynamic": "~7.1.0",
21 "@angular/router": "~7.1.0",
22 "core-js": "~2.5.4",
23 "rxjs": "~6.3.3",
24 "tslib": "^1.9.0",
25 "vendors": "file:dist/vendors/vendors-0.0.1.tgz",
26 "zone.js": "~0.8.26"
27 },
28 "devDependencies": {
29 "@angular-devkit/build-angular": "~0.11.0",
30 "@angular-devkit/build-ng-packagr": "~0.11.0",
31 "@angular/cli": "~7.1.2",
```

The 'vendors' dependency is highlighted with a yellow box.

Awesome! In this section, we learned how to deploy the Angular application as a standalone app and also as a composite app.

We also learned how to create a package of the Angular project that can be distributed and used in multiple Angular projects.

## Deploying Angular apps to GitHub Pages

In previous sections, we learned about deploying our standalone app and deploying the composite app to any server by exporting the compiled source files for the application.

In this section, we will learn how to deploy our Angular app to GitHub Pages.

Throughout the book, we have created many Angular projects and now it's time to host them, for free!

## Creating and deploying applications in GitHub Pages

GitHub Pages are websites for your projects hosted on GitHub. Did we say free? Of course, the GitHub Pages are free! Just edit, push, and view the changes live on your free website.

Let's take a look at how to create and host our application on GitHub Pages step by step:

1. Let's get started by installing Angular CLI using the `npm install` command:

```
npm install @angular/cli
```

2. Upon completion of the command, it's time to create a new Angular project. Let's call it `deploying-angular`:

```
ng new deploying-angular
```

Once the command is executed successfully, we should see the following screenshot:

The screenshot shows a Visual Studio Code window titled "deploying-angular - Visual Studio Code". The sidebar on the left shows the project structure with "DEPLOYING-ANGULAR" expanded, containing "deploying-angular", "node\_modules", and "package-lock.json". The main editor area is currently empty. At the bottom of the terminal pane, there is output from the command line:

```
+ angular/cli@7.1.2
added 292 packages from 178 contributors and audited 16181 packages in 66.045s
Found 0 vulnerabilities
```

Below this, another terminal window shows the execution of the `ng new` command:

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [http://sass-lang.com]
CREATE deploying-angular/angular.json (3958 bytes)
CREATE deploying-angular/package.json (1316 bytes)
CREATE deploying-angular/README.md (1033 bytes)
CREATE deploying-angular/tsconfig.json (435 bytes)
CREATE deploying-angular/tslint.json (2824 bytes)
CREATE deploying-angular/.editorconfig (246 bytes)
CREATE deploying-angular/.gitignore (576 bytes)
```

3. Now it's time to initiate a Git repository. We can do that by executing the following command:

```
git init
```

4. Upon successful execution, you will see the repository initialized or, in the following case, if a repository already exists, then it will be reinitialized as follows:

```
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/tslint.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in tsconfig.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in tslint.json.
The file will have its original line endings in your working directory.
Successfully initialized git.

D:\book_2\chapter15\deploying-angular>cd deploying-angular

D:\book_2\chapter15\deploying-angular\deploying-angular>git init
Reinitialized existing Git repository in D:/book_2/chapter15/deploying-angular/.git/

D:\book_2\chapter15\deploying-angular\deploying-angular>
```

5. Feel free to make any changes to app.component.html or any files that you would want to modify. Then, once you are ready to deploy, first commit the code/changes by executing the commit Git command. We can also pass the -m meta flag and add a message to the commit:

```
git commit -m "deploying angular"
```

6. Next, we need to set the origin to the repository. The following command sets the remote origin to the repository:

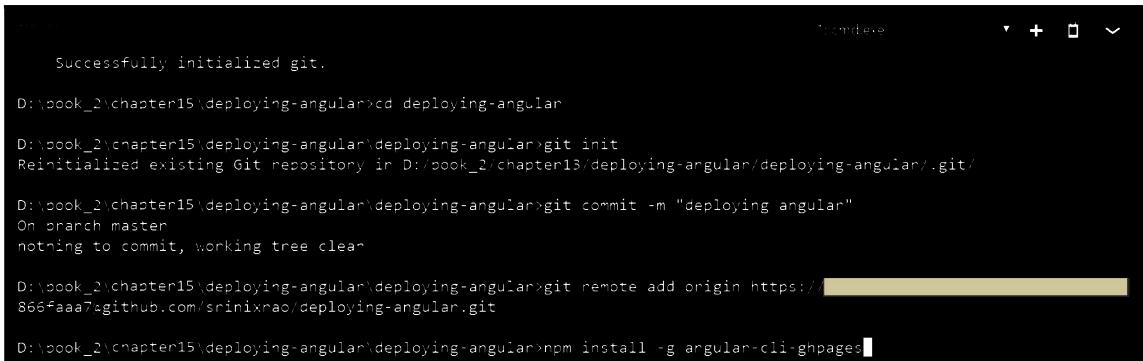
```
git remote add origin
https://<token>@github.com/<username>/<repo-name>
```

All right. All set.

7. Now, the superpowers come in. To deploy your Angular app to GitHub directly, we will need to install a package called angular-cli-ghpages. This is an official distribution to deploy Angular apps to GitHub Pages directly:

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

This is the output we will get up on running the preceding code:



```
Successfully initialized git.
D:\book_2\chapter15\deploying-angular>cd deploying-angular
D:\book_2\chapter15\deploying-angular\deploying-angular>git init
Reinitialized existing Git repository in D:/book_2/chapter15/deploying-angular/.git/
D:\book_2\chapter15\deploying-angular\deploying-angular>git commit -m "deploying angular"
On branch master
nothing to commit, working tree clean
D:\book_2\chapter15\deploying-angular\deploying-angular>git remote add origin https://[REDACTED]
866faaa7.github.com/srinixrao/deploying-angular.git
D:\book_2\chapter15\deploying-angular\deploying-angular>npm install -g angular-cli-ghpages
```

Now that we have `angular-cli-ghpages` installed, it's time to build our application and get the compiled source files.

8. Let's run the `ng build` command with the `--prod` meta flag and also set `--base-href`:

```
ng build --prod --base-href
"https://<username>.github.io/deploying-angular"
```



The `--base-href` flag is pointing to the source repository on GitHub. You will need to register with GitHub and get your authorization token in order to host your application.

9. Here is the `base href` URL, which is the author's GitHub home page, and the corresponding `deploying-angular` repository:

```
ng build --prod --base-href
"https://srinixrao.github.io/deploying-angular"
```

- Once we build the Angular application, we will see that the compiled source code is generated under `dist/<defaultProject>` -`<defaultProject>`. The compiled source is usually the folder name that we specify as the application name:

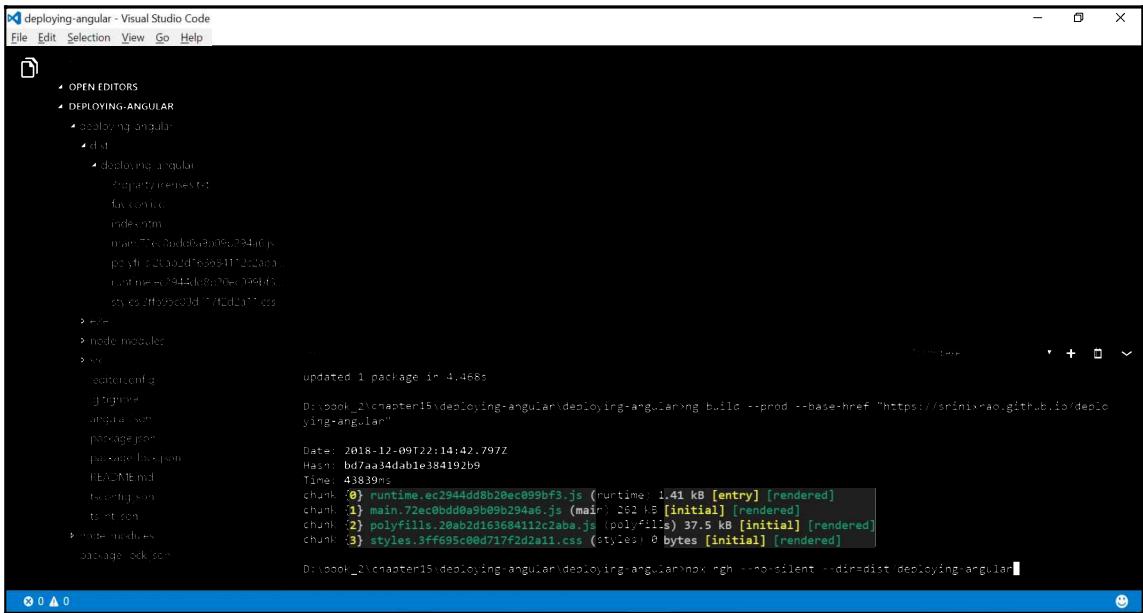
The screenshot shows the file structure of a deployed Angular application in Visual Studio Code. The project is named "deploying-angular". Inside the "dist" folder, there is a "deploy" folder which contains the compiled files. The terminal window shows the command "ng build --prod --base-href "https://srini-rao.github.io/deploy-angular-angular"" being run, and the output shows the creation of several chunks of code.

```
+ angular-cli-ghpages@0.5.0
updated 1 package in 4.468s
D:\book_2\chapter15\deploying-angular\deploying-angular\ng build --prod --base-href "https://srini-rao.github.io/deploy-angular-angular"
Date: 2018-12-09T22:14:42.797Z
Hash: bd7aa34dab1e3b419cb9
Time: 43839ms
chunk {0} runtime.ec2944dd8b28ec099bf3.js (runtime) 1.41 kB [entry] [rendered]
chunk {1} main.72ec0bddd9a9b9b294a6.js (main) 262 kB [initial] [rendered]
chunk {2} polyfills.20ab2d163684112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
```

- Now that we have our compiled files generated, it's time to deploy the application to GitHub Pages. We do this by running the `npx ngh --no-silent` command:

```
npx ngh --no-silent --dir=dist/deploying-angular
```

12. Remember that, optionally, we will need to mention the corresponding `dist` folder that we want to deploy:



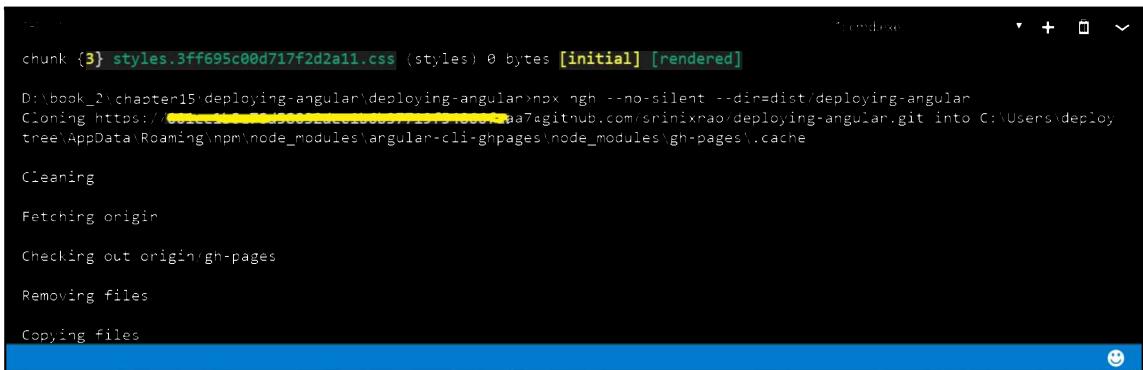
The screenshot shows the Visual Studio Code interface with the title bar "deploying-angular - Visual Studio Code". The left sidebar displays the project structure:

- OPEN EDITORS
- DEPLOYING-ANGULAR
- deploying-angular
  - dist
  - deploying-angular
    - index.html
    - main.js (file size 294.6KB)
    - polyfills.js (file size 37.5KB)
    - runtime.js (file size 294.6KB)
    - styles.css (file size 0B)
- node\_modules
- src

The main editor area shows the command line output of the deployment process:

```
D:\book_2\chapter15\deploying-angular>ng build --prod --base-href "https://srinixrao.github.io/deploying-angular"
updated 1 package in 4.48s
D:\book_2\chapter15\deploying-angular>ng build --prod --base-href "https://srinixrao.github.io/deploying-angular"
package.json
Date: 2018-12-09T22:14:42.797Z
Hash: bd7aa34dab1e384192b9
Time: 43839ms
chunk {0} runtime.ec2944dd8b28ec899bf3.js (runtime: 1.41 kB [entry] [rendered])
chunk {1} main.72ec0bdda9ab9b294a6.js (main) 292 kB [initial] [rendered]
chunk {2} polyfills.20ab2d16368a4112c2aba.js (polyfills) 37.5 kB [initial] [rendered]
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]
D:\book_2\chapter15\deploying-angular>gh-pages -no-silent --dir=dist deploying-angular
```

13. Upon successful execution of the command, the package we installed for deploying the Angular application to GitHub Pages will run the required jobs, such as cleaning, fetching the origin, checking out the code, and, finally, pushing the latest code to the repository, and will then be ready to host in GitHub Pages:



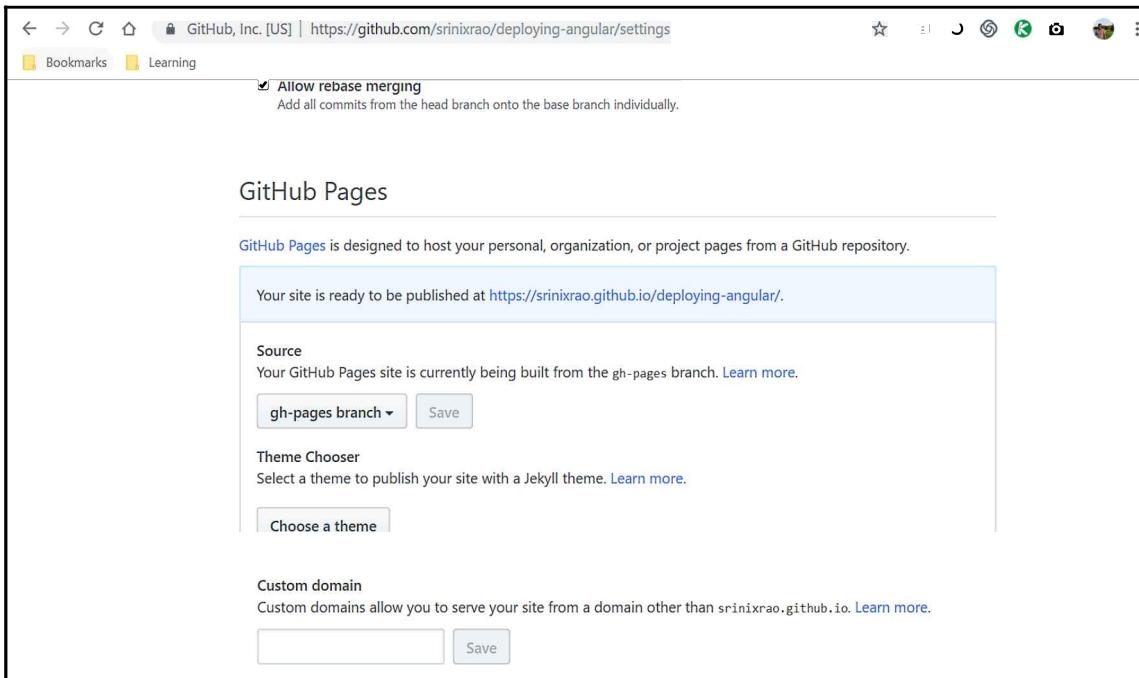
The screenshot shows a terminal window with the following output:

```
chunk {3} styles.3ff695c00d717f2d2a11.css (styles) 0 bytes [initial] [rendered]

D:\book_2\chapter15\deploying-angular>gh -no-silent --dir=dist/deploying-angular
Cloning https://github.com/srinixrao/deploying-angular.git into C:\Users\deploytree\AppData\Roaming\npm\node_modules\angular-cli-ghpages\node_modules\gh-pages\.cache

Cleaning
Fetching origin
Checking out origin\gh-pages
Removing files
Copying files
```

- Once the commands are executed, navigate to your GitHub account and click on **Settings** under the repository. You will see the site published to the URL:



- Click on the link displayed under the repository and we should see that our app is up and running!

Congratulations! We just published our first Angular application to GitHub Pages:



In the preceding series of steps, we learned how to deploy our Angular application to GitHub Pages. In more realistic scenarios, we will also need to deploy the APIs or backend services to our server. We can do that by deploying our APIs to either Firebase or self-hosted servers.

Now, go ahead and just repeat the same for all the projects and applications created so far.

# Summary

Deploying an application holds extreme importance: all our hard work of developing will show up once the site is alive.

Deploying Angular applications is pretty straightforward if you generate the required compiled source code, and, with latest versions of Angular, the AOT compilation defaults to any build generated with the `--prod` meta flag. We learned about the importance of AOT and how critical it is to have it for overall application performance and security. We learned to deploy a standalone Angular application and also composite Angular applications with multiple projects, libraries, and components.

Finally, we learned about deploying our Angular application to GitHub Pages using the official `angular-cli-ghpages` package.

That brings us to the conclusion of our last chapter in this book. We have come a long way in our learning journey, from understanding the basics of the TypeScript language to learning how to build our Angular applications by implementing the Angular framework's components, routing systems, directives, pipes, forms, backend services, and much more.

We also learned about implementing various CSS frameworks, such as Bootstrap, Angular Material, and Flex layout with our Angular applications. Additionally, we learned how to design and make our application's UI much more appealing and interactive.

We explored unit testing using the Jasmine and Karma frameworks, which ensures that our applications are well-tested and are rock-solid implementations.

We have also implemented the user authentication mechanism using Auth0 and Firebase as part of learning Angular Advanced topics. Finally, we covered the deployment of Angular applications.

That is a 360-degree overview of all aspects of application development using the Angular framework. We hope you now feel empowered to build world-class products using the Angular framework.

We wish you all the best and look forward to hearing about your success stories soon.

Good luck! Onward and upward.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## Getting Started with Web Components

Prateek Jadhwan

ISBN: 978-1-83864-923-4

- Understand Web Component design, specifications, and life cycle
- Create single-page applications using Web Components
- Enable reusability and customization for your UI components
- Implement Web Components in your web apps using Polymer and Stencil libraries
- Build powerful frontend components from scratch and deploy them on the web
- Design patterns and best practices to integrate Web Components into your existing web application



## **Flask Framework Cookbook, Second Edition**

Shalabh Aggarwal

ISBN: 978-1-78995-129-5

- Explore web application development in Flask, right from installation to post-deployment stages
- Make use of advanced templating and data modeling techniques
- Discover effective debugging, logging, and error handling techniques in Flask
- Integrate Flask with different technologies such as Redis, Sentry, and MongoDB
- Deploy and package Flask applications with Docker and Kubernetes
- Design scalable microservice architecture using AWS LambdaContinuous integration and Continuous deployment

## **Leave a review - let other readers know what you think**

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

## @

`@Component` annotation 186  
`@Component` decorator, properties  
  about 187  
  selector 187  
  styles 188, 189  
  stylesUrls 188, 189  
  template 187  
  templateUrl 187  
`@NgModule` decorator  
  properties 190, 191

## A

access modifiers  
  `private` 62  
  `protected` 62  
  `public` 62  
ahead-of-time (AOT) compilation  
  about 421  
  benefits 421  
alert components  
  reference link 108  
Angular 1.x  
  releases 8  
Angular 2 8  
Angular applications  
  ahead-of-time (AOT) compilation 421  
  architecting 184  
  architecture 184  
  Base Href 143  
  basic concepts 143  
  compilation options 420  
  component responsibilities 186  
  components, breaking up into sub-components 185  
  creating, in GitHub Pages 444, 445, 446, 447,

449, 450  
CRUD operations, overview 323  
deploying 419  
deploying, in GitHub Pages 444, 445, 446, 447, 449, 450  
deploying, to GitHub Pages 443  
fundamental concepts 320  
just-in-time (JIT) compilation 420  
location, specifying for rendering component templates 157  
NoSQL databases, concept 322, 323  
observables 322  
RouterLink 144  
RouterLinkActive 144  
routes, configuring for 145, 146, 147  
routing, strategies 158, 159  
running 157, 158  
shell, creating with CLI 138, 139, 140, 141, 142, 143  
strongly typed languages 321  
TypeScript interfaces 321, 322  
writing 14  
Angular CLI 12  
Angular components  
  testing 362, 363, 364, 365, 366  
Angular forms  
  about 289  
  data, submitting 306, 308  
  model-driven forms 295  
  validations 301  
Angular HTTP  
  integrating, with backend APIs 338, 339, 340, 341, 342, 343, 345, 346, 348, 349, 350  
  integrating, with Google Firebase 351, 352, 353, 354, 355, 357  
Angular HttpClient 331, 332, 333  
Angular Material

**about** 242, 243  
**buttons** 265, 267, 268  
**cards and layouts** 254  
**components** 247, 248, 249  
**data tables** 273  
**form controls** 261, 262, 263, 264  
**indicators** 265, 267, 268, 269  
**installing** 243, 244, 245, 247  
**modals** 270, 271, 272  
**navigation components** 250  
**popups** 270, 271, 272  
**URL** 242  
**Angular project**  
    packing, as npm package 440, 441, 442, 443  
**Angular release schedule**  
    reference link 9  
**Angular routing**  
    testing 374, 375, 376, 377, 378  
**Angular services**  
    testing 378, 380, 381, 382, 383, 384  
**Angular test automation** 362  
**Angular**  
    about 9  
    basics 18  
    components 10, 18, 19, 20  
    custom directives 212  
    dependency injection (DI) 11  
    directives 10  
    event binding 29  
    evolution 8, 9  
    facts 11  
    interpolation 24, 25  
    modules 10  
    property binding 27, 28  
    routing 137, 138  
    services 11  
    styling 26  
    templates 10  
    templating 26  
    URL 8  
**AngularJS**  
    URL 8  
**annotated photo album**  
    about 30  
**wireframes** 31

**annotations**  
    **@Component** 186  
    **about** 186  
**APIs**  
    **building, for ListingApp application** 324, 325, 326, 327  
**arrays** 51, 52  
**attribute directives**  
    about 201  
    reference 202  
**Auth0**  
    **user authentication, implementing** 407, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418

## B

**backend services**  
    **Angular HTTP, integrating with backend APIs** 338, 339, 340, 341, 342, 343, 345, 346, 348, 349, 350  
    **Angular HTTP, integrating with Google Firebase** 351, 352, 353, 354, 355, 357  
    **integrating** 337  
**Base Href** 143  
**Bootstrap form classes**  
    about 277, 279, 280  
    **inline form** 283, 284  
    **readonly attribute** 282  
    **sizing attribute** 281, 282  
**Bootstrap form**  
    about 276  
    **disabled attribute** 285, 286  
    **grid classes, using** 284, 285  
    **help text** 287  
    **input element, displaying as plain text** 288, 289  
**Bootstrap navigation bar** 154, 156  
**Bootstrap, components**  
    about 106  
    **alert components** 108  
    **button components** 106, 107  
    **modal components** 109  
    **Navbar component** 109  
**Bootstrap, responsive grid system**  
    **column** 102, 103  
    **container** 98, 99  
    **row** 99, 100, 101, 102

**viewport sizes** 103, 104, 105  
**Bootstrap**  
about 93, 94, 218  
crash course 93  
example application 95  
installing 95, 96  
motivation 94, 95  
responsive grid system 97, 98  
**built-in data types** 49  
**built-in directives**  
about 204  
**NgCase** 205  
**NgClass** 209  
**NgDefault** 205  
**NgIf** 204  
**NgNonBindable** 210  
**NgStyle** 207, 208  
**NgSwitch** 205  
**built-in functions, Sass**  
reference link 89  
**built-in pipes** 213, 215

## C

**cards and layout, Angular Material**  
accordions 258  
expansion panels 259  
list, with dividers 257  
lists 256  
material cards 255, 256  
navigation lists 257, 258  
steppers 260  
tabs 261  
**carousel example, NGB**  
reference link 234  
**child routes** 148, 149  
**chips** 268  
**Chromium Project** 43  
**classes** 59, 60  
**clear**  
using 165  
**Cloud Firestore**  
reference link 328  
**code minification** 317  
**collapse component**  
about 227

**declarations** 230  
**importations** 230  
**parent component** 227  
**command-line interface (CLI)** 11, 243  
**Compass**  
about 78  
URL 79  
**compilation**  
versus transpilation 45  
**component interface** 194, 195  
**components**  
about 10, 18, 19, 20  
associating 172  
implementing 196, 197  
layout techniques 162, 163  
**composite Angular applications**  
deploying 427  
**const keyword** 47  
**content projection**  
about 191, 192  
multiple sections, projecting 192  
**Create Listing wireframe** 176, 177, 178  
**CRUD** 150  
**CRUD operations**  
create 323  
delete 323  
overview 323  
read 323  
update 323  
**CSS Grid** 166  
**custom directives**  
about 212  
creating 390, 391, 392, 393  
**custom form validations** 393, 394, 395  
**custom pipes** 216

## D

**data binding**  
with NgModel directive 211  
**data types** 47  
**decorators** 66  
**default exports** 68, 69  
**dependency injection (DI)**  
about 11, 190, 309, 310, 311, 312, 313, 314,  
378

**testing** 378  
**design principles** 30  
**destructuring** 64, 65  
**development environment, Angular application**  
  **files location** 15  
  **to do list application, serving up** 16, 17  
  **to-do list application, generating** 15  
**development environment**  
  **setting up** 12, 14  
  **using** 14  
**directives**  
  **about** 10, 200, 201  
  **attribute directives** 201  
  **built-in directives** 204  
  **custom directives** 213  
  **structural directives** 202  
  **testing** 367, 368, 369, 370, 371, 372, 373

## E

**ECMAScript 2015** 44  
**Edit Listing wireframe** 179, 180  
**ES6**  
  **const keyword** 47  
  **let keyword** 46, 47  
**event binding** 29, 212

## F

**Firebase**  
  **about** 398  
  **reference link** 398  
  **user authentication, implementing** 398, 399,  
    401, 402, 404, 405, 406  
**Flex-Layout API**  
  **about** 170, 171  
  **reference link** 171  
**Flex-Layout**  
  **about** 161, 167  
  **advantages** 168  
  **integrating** 168, 169  
  **key points** 167  
  **reference link** 167  
**FlexBox CSS**  
  **about** 166  
  **features** 166, 167  
**FlexBox**

**design strategies** 171, 172  
**float**  
  **using** 164, 165  
**for-of loop** 65  
**forms** 277

## G

**GitHub Pages**  
  **Angular application, creating** 444, 445, 446,  
    447, 449, 450  
  **Angular application, deploying in** 444, 446, 447,  
    449, 450  
  **Angular application, deploying to** 443  
**Google Firebase**  
  **Angular HTTP, integrating** 351, 352, 353, 354,  
    355, 357  
**Google Firestore database** 328, 329, 330, 331

## H

**hard delete** 335  
**Heroku**  
  **URL** 79  
**HTML input elements** 277  
**HTTP DELETE method** 335  
**HTTP GET method** 333  
**HTTP POST method** 334  
**HTTP PUT method** 334  
**HTTP requests**  
  **testing** 384, 385, 387, 388, 389  
**HTTP responses**  
  **testing** 384, 385, 387, 388, 389  
**HTTP verbs** 332, 333  
**HTTP**  
  **via promises** 335, 336

## I

**inheritance** 63, 64  
**interfaces**  
  **about** 61, 62  
  **generating** 314, 315, 316  
**interpolation** 24, 25  
**Inversion of Control (IoC)** 11, 310  
**ISBN** 394

## J

**Jasmine** 359, 360, 361  
**JavaScript events**  
  about 43  
  Chromium Project 43  
  ECMAScript 2015 44  
  frameworks 44  
**JavaScript runtime environment** 51  
**JavaScript-powered Bootstrap components**  
  examples 218  
**JavaScript**  
  frameworks 44  
  with TypeScript 42  
**jQuery**  
  reference link 219  
**JSON** 50, 51  
**just-in-time (JIT) compilation** 420

## K

**Karma** 359, 361  
**Koala**  
  about 82  
  URL 81

## L

**Lean UX design process**  
  reference link 7  
**let keyword** 46, 47  
**life cycle hooks**  
  about 193  
  ngOnChanges 193  
  ngOnDestroy 194  
  ngOnInit 193  
**LinkedIn**  
  URL 79  
**Listing Carousel, implementation**  
  about 115  
    Create Listing page 127  
    Edit listing screen 127  
  interim web server, installing 115  
  listings 124, 126  
  login screen 122, 123  
  signup page 120, 121  
  Welcome page 115, 117, 118, 120

**Wireframes collection** 129, 130, 131, 132, 133, 134

## Listing Carousel

  about 73, 110  
  analysis 112, 113  
  concept 111  
  game plan 74, 75, 76, 77  
  idea generation 111  
  requirement gathering 113, 114  
  wireframes 114

## ListingApp application

  APIs, building 324, 325, 326, 327  
  database or fake API setup 320  
  Google Firestore database 328, 329, 330, 331  
  overview 319, 320  
  services or middleware layer 320  
  technical requisites 324  
  UI layer 319

## M

**maps** 53, 54, 55  
**modal components**  
  reference link 110  
**modal**  
  declarations 233  
  importations 233  
**model-driven forms**  
  about 295  
  cons 296  
  modules 297  
  pros 296  
  registration form example 298, 299, 301  
  validations, adding 304, 305, 306  
**module**  
  versus NgModule 190  
**modules** 10, 68  
**Mongo** 14  
**multiple Angular applications**  
  creating 428, 430, 431, 432, 433, 435, 436, 437, 438, 439  
  deploying 428, 430, 431, 432, 433, 435, 436, 437, 438, 439

## N

Navbar components  
  reference link 109  
navigation components, Angular Material  
  custom Material menus 251, 252  
  custom sidebar menus 253, 254  
  with schematics 250  
NGB carousel component 234  
NGB carousel component class 234  
NGB carousel component template 235, 236  
NGB collapse component  
  class 228  
  template 229, 230  
NGB modal component 230, 231  
NGB modal component class 231  
NGB modal component template 232  
NGB widgets  
  about 226  
  carousel component 234  
  collapse component 227  
  modal dialog windows 230, 231  
  reference link 226  
NGB  
  advantages 222, 223  
  documentation link 219  
  implementing, into example application 236, 237  
  installing 219, 220, 221  
  integrating 219  
  playground component class, creating 224, 225  
  playground CSS file, creating 225  
  playground directory, creating 224  
  playground menu item, creating 226  
  playground template file, creating 225  
  playground, creating for 223  
NgCase directive 205, 207  
NgClass directive 209  
NgDefault directive 205, 206  
NgFor directive  
  about 202  
  index value, accessing of iteration 203  
NgIf directive 204  
NgModel directive  
  using, for data binding 211  
NgModule

versus module 190  
NgModules 10  
NgNonBindable directive 210  
NgStyle directive 207, 208  
NgSwitch directive 205, 206  
node package manager (npm)  
  about 12  
  reference link 221  
  using 141  
NoSQL databases  
  concept 322, 323  
npm package  
  Angular project, packing as 440, 441, 442, 443

## O

objects 49  
observables 322

## P

paper prototyping 38, 39  
parameterized routes 147, 148  
pipes  
  about 213  
  applying 214  
  built-in pipes 213  
playground component class  
  creating 224, 225  
playground CSS file  
  creating 225  
playground directory  
  creating 224  
playground menu item  
  creating 226  
playground template file  
  creating 225  
Popper.js  
  reference link 219  
positioning 165  
primitive data types 48  
promises 67  
property binding 27, 28, 212

## R

reactive extensions (RxJS) 322  
reactive forms

**about** 295  
**registration form example** 298, 299, 301  
**validations, adding** 304, 305, 306

**route configuration**  
    **completing** 152, 154

**route guards**  
    **about** 149  
    **implementing** 150, 151

**router links directives** 154, 156

**RouterLink** 144

**RouterLinkActive** 144

**routes**  
    **child routes** 148, 149  
    **configuring, for Angular application** 145, 146, 147  
    **parameterized routes** 147, 148

**routing**  
    **in Angular** 137, 138

**Ruby**  
    **installing** 80

**S**

**sample project**  
    **about** 29  
    **annotated photo album** 30  
    **design principles** 30  
    **wireframes** 31

**Sass Meister**  
    **URL** 81

**Sass**  
    **about** 78  
    **built-in functions** 90, 91  
    **crash course** 77  
    **custom functions** 91, 92  
    **extend** 86  
    **features** 82  
    **imports** 85  
    **installing** 81  
    **mathematical operations** 84  
    **mixins** 87, 88, 89  
    **nesting** 83  
    **offline tools** 81  
    **online tools** 81  
    **placeholders** 90  
    **styles** 79, 80

**syntax** 82  
**URL** 78  
**variables** 83, 84

**Search Engine Optimization (SEO)** 12

**services**  
    **about** 11  
    **generating** 314, 315, 316

**set** 55, 56, 57

**Single Page Applications (SPAs)** 8

**single-page applications**  
    **advantages** 397  
    **building** 396

**soft delete** 335

**StackBlitz**  
    **URL** 14

**standalone Angular application**  
    **deploying** 421, 422, 423, 424, 425, 426, 427

**strongly typed languages** 321

**structural directives**  
    **about** 202  
    **NgFor** 202  
    **reference link** 203

**styling** 26

**T**

**table** 164

**template literals** 188

**template strings** 65

**template-driven forms**  
    **about** 290  
    **cons** 291  
    **login form, building** 292, 294, 295  
    **modules** 291  
    **pros** 291  
    **validations, adding** 302

**templates** 10, 199

**templating** 26

**Test Driven Development (TDD)** 359

**testing frameworks**  
    **about** 359  
    **Jasmine** 360, 361  
    **Karma** 361

**The Sass Way**  
    **URL** 79

**to-do list application**

**code listing** 20, 21, 23, 24

**transpilation**

**versus compilation** 45

**TypedArray** 52, 53

**TypeScript interfaces** 321, 322

**TypeScript**

**advantages** 45

**transpilation, versus compilation** 45

**with JavaScript** 42

## U

**user authentication**

**implementing** 397

**modules** 397

**with Auth0** 407, 409, 410, 411, 412, 413, 414,  
        415, 416, 417, 418

**with Firebase** 398, 399, 401, 402, 404, 405,  
        406

**user experience (UX)** 238

**UX design rules of thumb** 238, 239

## V

**validations**

**adding, in model-driven forms** 304, 306

**adding, in reactive forms** 304, 306

**adding, in template-driven forms** 302, 303

**view encapsulation** 189

**Visual Studio Code**

**download link** 14

## W

**WeakMap** 55

**WeakSet** 57, 58

**wireframes, annotated photo album**

**about** 31

**album viewer** 37, 38

**album, creating** 35

**dashboard** 32, 33

**home page** 32

**image upload** 33

**photo album listing** 36

**photo listing** 35

**photo preparation** 34

**workbench** 37

**wireframes**

**about** 31, 173, 174

**implementing** 174, 175

**wireframing tools** 31

## X

**XAMPP**

**about** 419

**download link** 419