

EXPERT INSIGHT

Learning Angular

A practical guide to building web applications
with modern Angular

Forewords by:

Bonnie Brennan

Founder of TechStackNation.com

Pablo Deeleman

Frontend Architect at GitKraken

Fifth Edition

Aristeidis Bampakos

packt

Learning Angular

Fifth Edition

A practical guide to building web applications with modern Angular

Aristeidis Bampakos



Learning Angular

Fifth Edition

Copyright © 2024 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 author, 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.

Publishing Product Manager: Lucy Wan

Acquisition Editor – Peer Reviews: Jane D’Souza

Project Editor: Janice Gonsalves

Development Editor: Rebecca Youé

Copy Editor: Safis Editing

Technical Editor: Gaurav Gavas

Proofreader: Safis Editing

Indexer: Hemangini Bari

Presentation Designer: Ajay Patule

Developer Relations Marketing Executive: Deepak Kumar

First published: April 2016

Second edition: December 2017

Third edition: September 2020

Fourth edition: February 2023

Fifth edition: December 2024

Production reference: 1271224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83508-748-0

www.packt.com

Contributors

About the author

Aristeidis Bampakos has over 20 years of experience in the software development industry. He currently works as a web development team leader at Plex-Earth, specializing in the development of web applications using Angular. His career started as a C# .NET developer, but he saw the potential of web development and moved toward it in early 2011. He began working with AngularJS and in 2020 he was officially recognized as a **Google Developer Expert (GDE)** for Angular.

Aristeidis is passionate about helping the developer community learn and grow. His love for teaching has led him to become an award-winning author of the successful book titles *Learning Angular* and *Angular Projects*. He enjoys being speaking about Angular in meetups, conferences, and podcasts. He is also currently leading the effort to make Angular accessible to the Greek development community by maintaining the open-source Greek translation of the official Angular documentation.

This book is dedicated to all people around the globe that strive with mental health issues.

About the reviewers

Thomas Laforgue is a married father living in the French Alps. He is an Angular freelancer with more than 8 years of experience in the frontend world, particularly in Angular. He has been a **Google Developer Expert (GDE)** for over a year and is well known for his open-source project, Angular Challenges. This project features more than 50 challenges designed to help developers improve their Angular skills. He is passionate about frontend technology and open-source projects. Outside of work, he enjoys sports and board games.

Martina Kraus has been active in the world of web development since her early years and, over time, has become an expert in the field of web security. As an Application Security Engineer, she focuses on integrating security best practices into all phases of software development. In her role as an Angular **Google Developer Expert (GDE)**, she loves to spread knowledge about Angular and web security at national and international conferences, regularly organizes ngGirls events (free Angular workshops for women) and the German Angular conference NG-DE. She is currently working on a book titled *Authorization and Authentication for Web Developers: A Practical Guide*, where she aims to share her knowledge.

Forewords

Dear Reader,

The book you’re holding continues a journey of knowledge and discovery that began nearly a decade ago. The origins of *Learning Angular* date back to the summer of 2015. During that time, Packt Publishing, with whom I’d had several discussions over the years, approached me to write a book on any topic of my choosing that would appeal to the frontend web developer community.

In the summer of 2015, it was already well known that the Angular team at Google was working on a new version of its framework. This was not merely a continuation of what AngularJS had been up to that point, but a complete rewrite from scratch. AngularJS was showing signs of aging and facing criticism regarding its operability and performance. In contrast, libraries like React and Vue were gaining more acceptance, and their future appeared bright and promising. Angular faced the significant challenge of winning back developers’ hearts in a race it was late to enter—perhaps too late already.

With only that idea in mind, the task of writing a book seemed daunting, aggravated by the fact that there was no documentation available. In the summer of 2015, Angular was still in the alpha stage, and the only way to familiarize oneself with the framework’s mechanics was to read the team’s official blog, which dribbled out its posts, or to reverse-engineer code that changed radically every week with each new release.

Doubts abounded: Would the resulting book be accurate enough? Would it be embraced by the public given the expectations created? Would it stand the test of time? Despite these concerns, this was the crucial moment to author a book on an entirely new frontend technology. Ultimately, the first edition of *Learning Angular 2* (which later dropped the version number in favor of just *Learning Angular*) was released in May 2016, after much effort and over two dozen rewrites. I honestly thought that journey would end there: perhaps a couple of dozen books would be sold at most, it would receive some positive reviews, and probably many negative ones.

I doubted Angular itself would last much longer either; despite its beautifully crafted architecture, the framework had arrived late to the party and relied on principles the community intended to bury in favor of functional programming paradigms.

Nearly ten years later, I'm delighted to say my judgment was wrong. The collective effort put into this book has enabled thousands of developers worldwide to create wonderful projects, contributing to a better, more accessible world for everyone. *Learning Angular 2* became a success, and its subsequent editions have been no less successful.

Meanwhile, Angular has continued to evolve and has broken paradigms in its continuous pursuit of evolution. From signals to deferrable views, native server-side rendering, improved lightning-fast compilation tools, a revamped syntax, and an enhanced transition API, along with hundreds of major and minor additions, Angular has demonstrated an unparalleled commitment to the community and influenced the future path of our industry. Right after its inception, Angular was considered an ugly duckling in the industry. Now, it is the new white swan that once again sets the pace for the rest.

However, this poses a huge challenge: Can a book capture the greatness of Angular, help readers confidently initiate themselves in it, and remain accessible and engaging, all while competing with the comprehensive information on angular.dev, its official website? The answer is yes, as long as Aristeidis Bampakos is leading this endeavor.

Aristeidis has been the driving force behind this franchise's success and I owe him an infinite debt of gratitude. His perseverance in meeting the community's expectations, his enormous technical skill in deconstructing complex concepts, and his excellent narrative ability are the reasons why I consider the book you are now holding a powerful key that will open doors to a fascinating future for you and many others.

It is an honor to write this foreword and a privilege to have shared this journey with Aristeidis Bampakos and the Packt team over nearly a decade. The journey doesn't end here. It is now up to you, dear reader, to take the next steps, and this book will be your best guide.

Bon voyage.

Pablo Deeleman

Frontend Architect at GitKraken, and previous author of Learning Angular

Hello friends,

I am honored to introduce an exceptional book written by one of my all-time favorite Angular experts, Aristeidis Bampakos. He is an established bestselling author, a well-respected Angular Google Developer Expert, a principal enterprise architect, and an open-source author. Over the years, Aris has become a trusted figure in the Angular community, having not only mastered the framework, but also contributed directly to translations and other improvements. His dedication to the Angular ecosystem is reflected not only in his contributions, but also in his passion for helping others grow their knowledge and skills.

For those looking to level up their Angular expertise, this book offers a comprehensive yet approachable overview of the framework. Aris has a unique ability to break down complex concepts into digestible content, making learning Angular accessible and enjoyable for developers of all levels. Whether you're just beginning your journey or architecting a production app, this guide will undoubtedly help you advance your understanding of Angular.

Beyond his Angular contributions, Aris has been a beloved and influential leader in our Tech Stack Nation community from the very start. His contributions go beyond just code; he brings wisdom, humility, and a genuine passion for sharing knowledge. I encourage you to visit one of our live events, where Aris can often be found sharing his insights—not only as a brilliant teacher and author who never stops asking questions and learning new things, but also as a caring and supportive friend to us all.

In the constantly evolving world of open-source tech, resources we can trust are increasingly valuable, and it's even more valuable to have someone like Aris to guide us through the ever-changing landscape of Angular. I have no doubt that you'll find this book priceless, as I have found Aris' contributions to our community over the years.

Enjoy your journey through Angular! If you have questions or comments after reading, I encourage you to reach out to Aris, as he's super friendly. You can also stop by Tech Stack Nation and ask for him, I bet he'd love to meet you!

Miles of smiles,

Bonnie Brennan

Founder of TechStackNation.com, Enterprise Architect, and Angular GDE

Join us on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/LearningAngular5e>

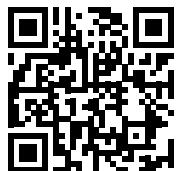


Table of Contents

Preface	xix
<hr/>	
Chapter 1: Building Your First Angular Application	1
Technical requirements	2
What is Angular?	2
Why choose Angular?	4
Cross-platform • 4	
Tooling • 4	
Onboarding • 5	
The usage of Angular worldwide • 5	
Setting up the Angular CLI workspace	6
Prerequisites • 6	
<i>Node.js</i> • 6	
<i>npm</i> • 7	
<i>Git</i> • 7	
Installing the Angular CLI • 7	
CLI commands • 8	
Creating a new project • 9	
The structure of an Angular application	12
Components • 13	
Bootstrapping • 13	
Template syntax • 14	

Angular tooling	16
Angular DevTools • 16	
VSCode Debugger • 20	
VSCode Profiles • 22	
<i>Angular Language Service</i> • 22	
<i>Material Icon Theme</i> • 24	
<i>EditorConfig</i> • 24	
Summary	25
Chapter 2: Introduction to TypeScript	27
Technical requirements	27
JavaScript essentials	28
Variable declaration • 28	
Function parameters • 31	
Arrow functions • 32	
Optional chaining • 33	
Nullish coalescing • 34	
Classes • 35	
Modules • 36	
What is TypeScript?	37
Getting started with TypeScript	39
Types • 41	
<i>String</i> • 42	
<i>Boolean</i> • 42	
<i>Number</i> • 42	
<i>Array</i> • 42	
<i>any</i> • 43	
<i>Custom types</i> • 43	
<i>Functions</i> • 44	
<i>Classes</i> • 45	

Interfaces • 48	
Generics • 50	
Utility types • 52	
Summary • 52	
Chapter 3: Structuring User Interfaces with Components	55
Technical requirements	55
Creating our first component	56
The structure of an Angular component • 56	
Creating components with the Angular CLI • 58	
Interacting with the template	60
Loading the component template • 60	
Displaying data from the component class • 62	
Controlling data representation • 63	
Class binding • 71	
Style binding • 72	
Getting data from the template • 73	
Component inter-communication	75
Passing data using an input binding • 75	
Listening for events using an output binding • 77	
Emitting data through custom events • 80	
Local reference variables in templates • 81	
Encapsulating CSS styling	82
Deciding on a change detection strategy	85
Introducing the component lifecycle	89
Performing component initialization • 90	
Cleaning up component resources • 91	
Detecting input binding changes • 93	
Accessing child components • 95	
Summary	96

Chapter 4: Enriching Applications Using Pipes and Directives	99
Technical requirements	99
Manipulating data with pipes	99
Building pipes	106
Sorting data using pipes • 106	
Passing parameters to pipes • 110	
Change detection with pipes • 112	
Building directives	113
Displaying dynamic data • 114	
Property binding and responding to events • 118	
Summary	120
Chapter 5: Managing Complex Tasks with Services	121
Technical requirements	122
Introducing Angular DI	122
Creating our first Angular service	124
Injecting services in the constructor • 126	
The inject keyword • 128	
Providing dependencies across the application	129
Injecting services in the component tree	133
Sharing dependencies through components • 133	
Root and component injectors • 138	
Sandboxing components with multiple instances • 139	
Restricting provider lookup • 145	
Overriding providers in the injector hierarchy	147
Overriding service implementation • 147	
Providing services conditionally • 149	
Transforming objects in Angular services • 151	
Summary	153

Chapter 6: Reactive Patterns in Angular	155
Technical requirements	155
Strategies for handling asynchronous information	156
Shifting from callback hell to promises • 156	
Observables in a nutshell • 160	
Reactive programming in Angular	162
The RxJS library	165
Creating observables • 166	
Transforming observables • 167	
Subscribing to observables	169
Unsubscribing from observables	172
Destroying a component • 172	
Using the async pipe • 174	
Summary	176
Chapter 7: Tracking Application State with Signals	177
Technical requirements	177
Understanding signals	178
Reading and writing signals	178
Computed signals	180
Cooperating with RxJS	182
Summary	185
Chapter 8: Communicating with Data Services over HTTP	187
Technical requirements	187
Communicating data over HTTP	188
Introducing the Angular HTTP client	189
Setting up a backend API	191
Handling CRUD data in Angular	192
Fetching data through HTTP • 192	

Modifying data through HTTP • 202	
<i>Adding new products</i> • 203	
<i>Updating product price</i> • 207	
<i>Removing a product</i> • 210	
Authentication and authorization with HTTP	214
Authenticating with a backend API • 214	
Authorizing user access • 216	
Authorizing HTTP requests • 218	
Summary	222
<hr/>	
Chapter 9: Navigating through Applications with Routing	223
<hr/>	
Technical requirements	224
Introducing the Angular router	224
Specifying a base path • 226	
Enabling routing in Angular applications • 226	
Configuring the router • 227	
Rendering components • 228	
Configuring the main routes	228
Organizing application routes	232
Navigating imperatively to a route • 233	
Using built-in route paths • 238	
Styling router links • 239	
Passing parameters to routes	240
Building a detail page using route parameters • 240	
Reusing components using child routes • 245	
Taking a snapshot of route parameters • 247	
Filtering data using query parameters • 248	
Binding input properties to routes • 250	
Enhancing navigation with advanced features	252
Controlling route access • 252	
Preventing navigation away from a route • 254	

Prefetching route data • 256	
Lazy-loading parts of the application • 259	
<i>Protecting a lazy-loaded route • 262</i>	
Summary	263
<hr/>	
Chapter 10: Collecting User Data with Forms	265
<hr/>	
Technical requirements	265
Introducing web forms	265
Building template-driven forms	267
Building reactive forms	271
Interacting with reactive forms • 271	
Creating nesting form hierarchies • 276	
Modifying forms dynamically • 278	
Using a form builder • 285	
Validating input in forms	288
Global validation with CSS • 288	
Validation in template-driven forms • 290	
Validation in reactive forms • 294	
Building custom validators • 297	
Manipulating form state	303
Updating form state • 303	
Reacting to state changes • 304	
Summary	306
Join Us on Discord	306
<hr/>	
Chapter 11: Handling Application Errors	307
<hr/>	
Technical requirements	307
Handling runtime errors	308
Catching HTTP request errors • 308	
Creating a global error handler • 312	
Responding to the 401 Unauthorized error • 315	

Demystifying framework errors	316
Summary	318
Chapter 12: Introduction to Angular Material	319
Technical requirements	319
Introducing Material Design	320
Introducing Angular Material	320
Installing Angular Material • 321	
Adding UI components • 324	
Theming UI components • 325	
Integrating UI components	329
Form controls • 330	
<i>Input</i> • 330	
<i>Select</i> • 335	
<i>Chips</i> • 337	
Navigation • 338	
Layout • 340	
<i>Card</i> • 341	
<i>Data table</i> • 344	
Popups and overlays • 350	
<i>Creating a confirmation dialog</i> • 350	
<i>Configuring dialogs</i> • 353	
<i>Getting data from dialogs</i> • 354	
<i>Displaying user notifications</i> • 355	
Summary	359
Chapter 13: Unit Testing Angular Applications	361
Technical requirements	362
Why do we need unit tests?	362
The anatomy of a unit test	363

Introducing unit tests in Angular	365
Testing components	366
Testing with dependencies • 370	
<i>Replacing the dependency with a stub • 371</i>	
<i>Spying on the dependency method • 375</i>	
<i>Testing asynchronous services • 378</i>	
Testing with inputs and outputs • 380	
Testing with a component harness • 383	
Testing services	385
Testing synchronous/asynchronous methods • 386	
Testing services with dependencies • 387	
Testing pipes	389
Testing directives	390
Testing forms	392
Testing the router	395
Routed and routing components • 395	
Guards • 398	
Resolvers • 401	
Summary	403
Chapter 14: Bringing Applications to Production	405
Technical requirements	406
Building an Angular application	406
Building for different environments • 408	
Building for the window object • 410	
Limiting the application bundle size	411
Optimizing the application bundle	412
Deploying an Angular application	415
Summary	416

Chapter 15: Optimizing Application Performance	417
Technical requirements	418
Introducing Core Web Vitals	418
Rendering SSR applications	422
Overriding SSR in Angular applications • 425	
Optimizing image loading	428
Deferring components	430
Introducing deferrable views • 430	
Using deferrable blocks • 431	
Loading patterns in @defer blocks • 437	
Prerendering SSG applications	440
Summary	441
Other Books You May Enjoy	445
Index	449

Preface

As Angular continues to reign as one of the top JavaScript frameworks, more developers are seeking out the best way to get started with this extraordinarily flexible and secure framework. *Learning Angular*, now in its fifth edition, will show you how you can use Angular to achieve cross-platform high performance with the latest web techniques, extensive integration with modern web standards, and integrated development environments (IDEs).

This book is especially useful for those new to Angular and will help you to get to grips with the bare bones of the framework needed to start developing Angular apps. You'll learn how to develop apps by harnessing the power of the Angular command-line interface (CLI), write unit tests, style your apps by following the Material Design guidelines, and finally, build them for production.

Updated for Angular 19, this new edition covers lots of new features and practices that address the current frontend web development challenges. You'll find new dedicated chapters on signals and optimization, as well as more on error handling and debugging in Angular, and new real-life examples. By the end of this book, you'll not only be able to create Angular applications with TypeScript from scratch, but also enhance your coding skills with best practices.

Who this book is for

This book is for web developers that want to get started with frontend development, and frontend developers that want to expand their knowledge of JavaScript frameworks. You'll need prior exposure to JavaScript, basic knowledge of the command line, and to be comfortable with using IDEs to get started with this book.

What this book covers

Chapter 1, Building Your First Angular Application

In this chapter, we set up the development environment by installing the Angular CLI and learn how to use schematics (commands) to automate tasks such as code generation and application building. We create a new simple application using the Angular CLI and build it. We also learn about some of the most useful Angular tools that are available in Visual Studio Code.

Chapter 2, Introduction to TypeScript

In this chapter, we learn what TypeScript is, the language that is used when creating Angular applications, and what the most basic building blocks are, such as types and classes. We take a look at some of the advanced types available and the latest features of the language.

Chapter 3, Structuring User Interfaces with Components

In this chapter, we learn how a component is connected to its template and use a decorator to configure it. We take a look at how components communicate with each other by passing data from one component to another using input and output bindings and learn about the different strategies to detect changes in a component. We also learn how to execute custom logic during the component lifecycle.

Chapter 4, Enriching Applications Using Pipes and Directives

In this chapter, we take a look at Angular's built-in pipes and we build our own custom pipe. We learn how to create directives and leverage them through an Angular application that demonstrates their use.

Chapter 5, Managing Complex Tasks with Services

In this chapter, we learn how the dependency injection mechanism works, create and use services in components into components, and how to create providers in an Angular application.

Chapter 6, Reactive Patterns in Angular

In this chapter, we learn what reactive programming is and how we can use observables in the context of an Angular application through the RxJS library. We also take a tour of all the common RxJS operators that are used in an Angular application.

Chapter 7, Tracking Application State with Signals

In this chapter, we learn the basic concepts of the Signals API and the rationale behind its use. We explore how to use signals for tracking the state of an Angular application. We also take a look at signals interoperability with RxJS and how they can play nicely together in a sample application.

Chapter 8, Communicating with Data Services over HTTP

In this chapter, we learn how to interact with a remote backend API and perform CRUD operations with data in Angular. We also investigate how to set additional headers to an HTTP request and intercept it before sending the request or upon completion.

Chapter 9, Navigating through Applications with Routing

In this chapter, we learn how to use the Angular router in order to activate different parts of an Angular application. We find out how to pass parameters through the URL and how to break an application into child routes that can be lazy loaded. We then learn how to guard against our components and how to prepare data prior to initialization of the component.

Chapter 10, Collecting User Data with Forms

In this chapter, we learn how to use Angular forms in order to integrate HTML forms into an application and how to set them up using FormGroup and FormControl. We track the interaction of the user in the form and validate input fields.

Chapter 11, Handling Application Errors

In this chapter, we learn how to handle different types of errors in an Angular application and learn about errors that come from the framework itself.

Chapter 12, Introduction to Angular Material

In this chapter, we learn how to integrate Google Material Design guidelines in to an Angular application using a library called Angular Material, developed by the Angular team. We take a look at some of the core components of the library and how to use them. We discuss the themes that are bundled with the library and how to install them.

Chapter 13, Unit Testing Angular Applications

In this chapter, we learn how to test Angular artifacts and override them in a test, what the different parts of a test are, and which parts of a component should be tested.

Chapter 14, Bringing Applications to Production

In this chapter, we learn how to use the Angular CLI to build and deploy an Angular application. We take a look at how to pass environment variables during the build and how to perform build optimizations prior to deployment.

Chapter 15, Optimizing Application Performance

In this chapter, we learn what **Core Web Vitals (CWV)** are and how they affect the performance of an Angular application. We explore three different ways to improve CWV metrics: how to render an application server-side, how to benefit from hydration, and how to optimize our images.

To get the most out of this book

You will need a version of Angular 19 installed on your computer, preferably the latest one. All code examples have been tested using Angular 19.0.0 on Windows, but they should work with any future release of Angular 19 as well.

We advise you to type the code for this book yourself or access the code via the GitHub repository (the link is in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at <http://www.packtpub.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen 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
- Ziipeg/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/Learning-Angular-Fifth-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781835087480>.

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 social media handles. For example; “Mount the downloaded WebStorm-10*.dmg disk image file as another disk in your system.”

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
      /etc/asterisk/cdr_mysql.conf
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes. For example: “Select **System info** from the **Administration** panel.”



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: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@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, <http://www.packtpub.com/submit-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@packtpub.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 <http://authors.packtpub.com>.

Reviews

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 packtpub.com.

Share your thoughts

Once you've read *Learning Angular, Fifth Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835087480>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

1

Building Your First Angular Application

Web development has undergone huge growth during the last decade. Frameworks, libraries, and tools have emerged that enable developers to build great web applications. Angular has paved the way by creating a framework focusing on application performance, development ergonomics, and modern web techniques.

Before developing Angular applications, we need to learn some basic but essential things to have a great experience with the Angular framework. One of the primary things we should know is what Angular is and why we should use it for web development. We will also take a tour in this chapter of Angular history to understand how the framework has evolved.

Another important but sometimes challenging introductory topic is setting up our development environment. It must be done at the beginning of a project and getting this right early can reduce friction as our application grows. Therefore, a large part of this chapter is dedicated to the **Angular CLI**, a tool developed by the Angular team that provides scaffolding and automation tasks in an Angular application, eliminating configuration boilerplate and enabling developers to focus on the coding process. We will use the Angular CLI to create our first application from scratch, get a feel for the anatomy of an Angular application, and take a sneak peek at how Angular works under the hood.

Working on an Angular project without help from development tools, such as an **Integrated Development Environment (IDE)**, can be painful. Our favorite code editor can provide an agile development workflow that includes compilation at runtime, static type checking, introspection, code completion, and visual assistance to debug and build our application. We will highlight some of the most popular tools in the Angular ecosystem in this chapter, such as **Angular DevTools** and **Visual Studio Code (VSCode)**.

To sum up, here are the main topics that we will explore in this chapter:

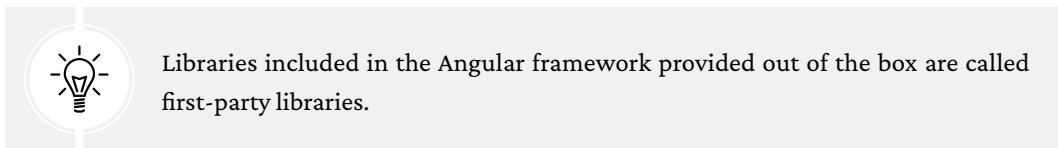
- What is Angular?
- Why choose Angular?
- Setting up the Angular CLI workspace
- The structure of an Angular application
- Angular tooling

Technical requirements

- **GitHub:** <https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition/tree/main/ch01>
- **Node.js:** <https://nodejs.org>
- **Git:** <https://git-scm.com>
- **VSCode:** <https://code.visualstudio.com>
- **Angular DevTools:** <https://angular.dev/tools/devtools>

What is Angular?

Angular is a web framework written in the **TypeScript** language and includes a CLI, a language service, a debugging tool, and a rich collection of first-party libraries.



Libraries included in the Angular framework provided out of the box are called first-party libraries.

Angular enables developers to build scalable web applications with TypeScript, a strict syntactic superset of JavaScript, which we will learn about in *Chapter 2, Introduction to TypeScript*.

The official Angular documentation can be found at <https://angular.dev>.



The official Angular documentation is the most up-to-date resource for Angular development. It's preferable to use it over other external resources while developing with Angular.

Google created Angular. The first version, 1.0, was released in 2012 and was called **AngularJS**. AngularJS was a JavaScript framework, and web applications built with it were written in JavaScript.

In 2016, the Angular team decided to make a revolutionary change in AngularJS. They collaborated with the TypeScript team at Microsoft and introduced the TypeScript language into the framework. The next version of the framework, 2.0, was written in TypeScript and rebranded as **Angular** with a different logo than AngularJS.

In 2022, Angular entered a new era of evolutionary advancements known as the *Angular Renaissance*. During that period, the framework picked up momentum in web development by introducing major innovations focused on enhancing the **Developer Experience (DX)** and optimizing application performance, such as:

- A simple and modern approach to authoring Angular applications
- Improved reactivity patterns to manage application state efficiently
- The integration of **Server-Side Rendering (SSR)** techniques to improve performance

A major milestone in the Angular Renaissance era was **Angular 17**, when the Angular team decided to rebrand the framework with a new logo and colors, reflecting the recent changes and setting the vision for future advancements.



In this book, we will cover **Angular 19**, the latest major *stable* version of the Angular framework. AngularJS reached the end of its life in 2022, and it is no longer supported and maintained by the Angular team.

Angular is based on the most modern web standards and supports all the evergreen browsers. You can find more details about the specific version support of each browser at <https://angular.dev/reference/versions#browser-support>.

In the following section, we will learn the benefits of choosing Angular for web development.

Why choose Angular?

The power of the Angular framework is based on the combination of the following characteristics:

- The main pillars of the framework:
 - Cross-platform
 - Incredible tooling
 - Easy onboarding
- The usage of Angular worldwide:
 - An amazing community
 - Battle-tested against Google products

In the following sections, we will examine each characteristic in more detail.

Cross-platform

Angular applications can run on different platforms: web, server, desktop, and mobile. Angular can run natively only on the web because it is a web framework; however, it is open-source and is backed by incredible tooling that enables the framework to run on the remaining three using the following tools:

- **Angular SSR**: Renders Angular applications server-side
- **Angular service worker**: Enables Angular applications to run as **Progressive Web Applications (PWAs)** that can execute in desktop and native mobile environments
- **Ionic/NativeScript**: Allows us to build mobile applications using Angular

The next pillar of the framework describes the tooling available in the Angular ecosystem.

Tooling

The Angular team has built two great tools that make Angular development easy and fun:

- **Angular CLI**: A command-line interface that allows us to work with Angular projects, from scaffolding to testing and deployment
- **Angular DevTools**: A browser extension that enables us to inspect and profile Angular applications from the comfort of our browser

The Angular CLI is the de facto solution to work with Angular applications. It allows the developer to focus on writing application code, eliminating the boilerplate of configuration tasks such as scaffolding, building, testing, and deploying an Angular application.

Onboarding

It is simple and easy to start with Angular development because when we install Angular, we also get a rich collection of first-party libraries out of the box, including:

- An Angular HTTP client to communicate with external resources over HTTP
- Angular forms to create HTML forms to collect input and data from users
- An Angular router to perform in-app navigations

The preceding libraries are installed by default when we create a new Angular application using the Angular CLI. However, they are only used in our application if we import them explicitly into our project.

The usage of Angular worldwide

Many companies use Angular for their websites and web applications. The website <https://www.madewithangular.com> contains an extensive list of those companies, including some popular ones.

Additionally, Angular is used in thousands of projects by Google and by millions of developers worldwide. The fact that Angular is already used internally at Google is a crucial factor in the reliability of the framework. Every new version of Angular is thoroughly tested in those projects before becoming available to the public. The testing process helps the Angular team catch bugs early and delivers a top-quality framework to the rest of the developer community.

Angular is backed and supported by a thriving developer community. Developers can access many available communities worldwide, online or locally, to get help and guidance with the Angular framework. On the other hand, communities help the Angular framework progress by sharing feedback on new features, testing new ideas, and reporting issues. Some of the most popular online communities are:

- **Tech Stack Nation:** The world's friendliest Angular study group that brings together Angular developers who are passionate about improving their confidence in building amazing Angular applications. Tech Stack Nation is a community where Angular developers can collaborate, learn from each other's expertise, and push the boundaries of what Angular can achieve. You can join Tech Stack Nation at <https://techstacknation.com>.

- **Angular Community Discord:** Angular's official Discord server that brings the incredible Angular community together. Everyone is welcome to join the community with the click of a button. It is the central location to connect Angular team members, **Google Developer Experts (GDEs)**, library authors, meetup groups, and anyone interested in learning the framework. You can join the Angular Community Discord server at <https://discord.gg/angular>.
- **Angular.love:** A community platform for Angular enthusiasts, supported by *House of Angular*, to facilitate the growth of Angular developers through knowledge-sharing initiatives. It started as a blog where experts published articles about Angular news, features, and best practices. Now, Angular.love also organizes in-person and online meetups, frequently featuring GDEs. You can join Angular.love at <https://angular.love>.

Now that we have seen what Angular is and why someone should choose it for web development, we will learn how to use it and build great web applications.

Setting up the Angular CLI workspace

Setting up a project with Angular can be tricky. You need to know what libraries to import and ensure that files are processed in the correct order, which leads us to the topic of scaffolding. Scaffolding is a tool to automate tasks, such as generating a project from scratch, and it becomes necessary as complexity grows and where every hour counts toward producing business value, rather than being spent fighting configuration problems.

The primary motivation behind creating the Angular CLI was to help developers focus on application building, eliminating the configuration boilerplate. Essentially, with a simple command, you should be able to initialize an application, add new artifacts, run tests, update applications, and create a production-grade bundle. The Angular CLI supports all of this using special commands called **schematics**.

Prerequisites

Before we begin, we must ensure that our development environment includes software tools essential to the Angular development workflow.

Node.js

Node.js is a JavaScript runtime built on top of Chrome's v8 JavaScript engine. Angular requires an active or maintenance **Long-Time Support (LTS)** version. If you have already installed it, you can run `node -v` on the command line to check which version you are running.



If you need to work with applications that use different Node.js versions or can't install the runtime due to restricted permissions, use **nvm**, a version manager for Node.js designed to be installed per user. You can learn more at <https://github.com/nvm-sh/nvm>.

npm

npm is a software package manager that is included by default in Node.js. You can check this out by running `npm -v` in the command line. An Angular application consists of various libraries, called *packages*, that exist in a central place called the *npm registry*. The npm client downloads and installs the libraries needed to run your application from the npm registry to your local computer.

Git

Git is a client that allows us to connect to distributed version-control systems, such as GitHub, Bitbucket, and GitLab. It is optional from the perspective of the Angular CLI. You should install it if you want to upload your Angular project to a Git repository, which you might want to do.

Installing the Angular CLI

The Angular CLI is part of the Angular ecosystem and can be downloaded from the npm package registry. Since it is used to create Angular projects, we must install it globally in our system. Open a terminal window and run the following command:

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



You may need elevated permissions on some Windows systems, so you should run your terminal as an administrator. Run the preceding command in Linux/macOS systems by adding the `sudo` keyword as a prefix to execute with administrative privileges.

The command that we used to install the Angular CLI uses the `npm` client, followed by a set of runtime arguments:

- `install` or `i`: Denotes the installation of a package
- `-g` or `--global`: Indicates that the package will be installed on the system globally
- `@angular/cli`: The name of the package to install

The Angular CLI follows the same version as the Angular framework, which in this book is 19. The preceding command will install the latest *stable* version of the Angular CLI. You can check which version you have installed by running `ng version` or `ng v` in the command line. If you have a different version than 19 after installing it, you can run the following command:

```
npm install -g @angular/cli@19
```

The preceding command will fetch and install the latest version of Angular CLI 19.

CLI commands

The Angular CLI is a command-line interface tool that automates specific tasks during development, such as serving, building, bundling, updating, and testing an Angular project. As the name implies, it uses the command line to invoke the `ng` executable file and run commands using the following syntax:

```
ng [command] [options]
```

Here, `[command]` is the name of the command to be executed, and `[options]` denotes additional parameters that can be passed to each command. To view all available commands, you can run the following:

```
ng help
```

Some commands can be invoked using an alias instead of the name. In this book, we cover the most common ones (the alias of each command is shown inside parentheses):

- `new (n)`: Creates a new Angular CLI workspace from scratch
- `build (b)`: Compiles an Angular application and outputs generated files in a predefined folder
- `generate (g)`: Creates new files that comprise an Angular application
- `serve (dev)`: Builds an Angular application and serves it using a pre-configured web server
- `test (t)`: Runs the unit tests of an Angular application
- `add`: Installs an Angular library in an Angular application
- `update`: Updates an Angular application to the latest Angular version

You can find more Angular CLI commands at <https://angular.dev/cli>.

Updating an Angular application is one of the most critical tasks from the preceding list. It helps us stay up to date by upgrading our Angular applications to the latest version.



Try to keep your Angular projects up to date because each new version of Angular comes packed with many exciting new features, performance improvements, and bug fixes.

Additionally, you can use the Angular upgrade guide, which contains tips and step-by-step instructions on updating your applications, at <https://angular.dev/update-guide>.

Creating a new project

Now that we have prepared our development environment, we can start creating our first Angular application. We will use the `ng new` command of the Angular CLI and pass the name of the application that we want to create as an option:

1. Open a terminal window, navigate to a folder of your choice, and run the command `ng new my-app`. Creating a new Angular application is a straightforward process. The Angular CLI will ask for details about the application we want to create so that it can scaffold the Angular project as best as possible.
2. Initially, it will ask if we want to enable Angular analytics:

Would you like to share pseudonymous usage data about this project with the Angular Team at Google under Google's Privacy Policy at <https://policies.google.com/privacy>. For more details and how to change this setting, see <https://angular.dev/cli/analytics>. (y/N)

The Angular CLI will ask this question once when we create the first Angular project and apply it globally in our system. However, we can change the setting later in a specific Angular workspace.

3. The next question is related to the styling of our application:

Which stylesheet format would you like to use?

It is common to use CSS to style Angular applications. However, we can use preprocessors like **SCSS** or **Less** to add value to our development workflow. In this book, we work with CSS directly, so accept the default choice, CSS, and press *Enter*.

4. Finally, the Angular CLI will prompt us if we want to enable SSR and **Static Site Generation (SSG)** in our application:

```
Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N)
```

SSR and SSG are concerned with improving the startup and load performance of an Angular application. We will learn more about them in *Chapter 15, Optimizing Application Performance*. For now, accept the default choice, No, by pressing *Enter*.

The process may take some time, depending on your internet connection. During this time, the Angular CLI downloads and installs all necessary packages and creates default files for your Angular application. When finished, it will have created a folder called `my-app`. The folder represents an Angular CLI workspace that contains a single Angular application called `my-app` at the root level.

The workspace contains various folders and configuration files that the Angular CLI needs to build and test the Angular application:

- `.vscode`: Includes VSCode configuration files
- `node_modules`: Includes installed npm packages that are needed to develop and run the Angular application
- `public`: Contains static assets such as fonts, images, and icons
- `src`: Contains the source files of the application
- `.editorconfig`: Defines coding styles for the default editor
- `.gitignore`: Specifies the files and folders that Git should not track
- `angular.json`: The main configuration file of the Angular CLI workspace
- `package.json` and `package-lock.json`: Provide definitions of npm packages, along with their exact versions, which are needed to develop, test, and run the Angular application
- `README.md`: A README file that is automatically generated from the Angular CLI
- `tsconfig.app.json`: A TypeScript configuration that is specific to the Angular application
- `tsconfig.json`: A TypeScript configuration that is specific to the Angular CLI workspace
- `tsconfig.spec.json`: A TypeScript configuration that is specific to unit tests of the Angular application

As developers, we should only care about writing the source code that implements features for our application. Nevertheless, having basic knowledge of how the application is orchestrated and configured helps us better understand the mechanics and ways to intervene if necessary.

Navigate to the newly created folder and start your application with the following command:

```
ng serve
```



The Angular CLI compiles the Angular project and starts a web server that watches for changes in project files. This way, whenever you change your application code, the web server rebuilds the project to reflect the new changes.

After compilation has been completed successfully, you can preview the application by opening your browser and navigating to `http://localhost:4200`:

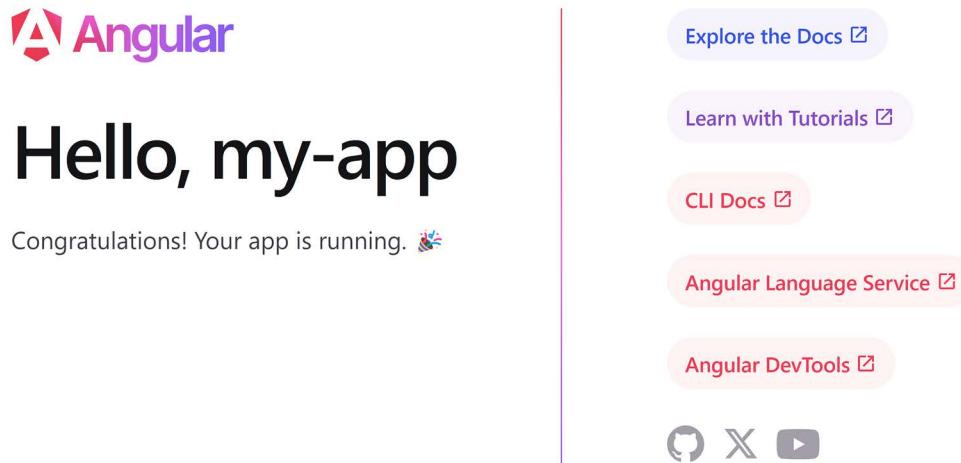


Figure 1.1: Angular application landing page

Congratulations! You have created your first Angular CLI workspace. The Angular CLI created a sample web page that we can use as a reference to build our project. In the next section, we will explore the main parts of our application and learn how to modify this page.

The structure of an Angular application

We will take the first intrepid steps in examining our Angular application. The Angular CLI has already scaffolded our project and done much of the heavy lifting for us. All we need to do is fire up our favorite IDE and start working with the Angular project. We will use VSCode in this book, but feel free to choose any editor you are comfortable with:

1. Open VSCode and select **File | Open Folder...** from the main menu.
2. Navigate to the `my-app` folder and select it. VSCode will load the associated Angular CLI workspace.
3. Expand the `src` folder.

When we develop an Angular application, we'll likely interact with the `src` folder. It is where we write the code and tests of our application. It contains the following:

- `app`: All the Angular-related files of the application. You interact with this folder most of the time during development.
- `index.html`: The main HTML page of the Angular application.
- `main.ts`: The main entry point of the Angular application.
- `styles.css`: CSS styles that apply globally to the Angular application. The extension of this file depends on the stylesheet format you choose when creating the application.

The `app` folder contains the actual source code we write for our application. Developers spend most of their time inside that folder. The Angular application that was created automatically from the Angular CLI contains the following files:

- `app.component.css`: Contains CSS styles specific to the sample page. The extension of this file depends on the stylesheet format you choose when creating the application.
- `app.component.html`: Contains the HTML content of the sample page.
- `app.component.spec.ts`: Contains unit tests for the sample page.
- `app.component.ts`: Defines the *presentational logic* of the sample page.
- `app.config.ts`: Defines the configuration of the Angular application.
- `app.routes.ts`: Defines the routing configuration of the Angular application.



The filename extension `.ts` refers to TypeScript files.

In the following sections, we will learn how Angular orchestrates some of those files to display the sample page of the application.

Components

The files whose names start with `app.component` constitute an **Angular component**. A component in Angular controls part of a web page by orchestrating the interaction of the presentational logic with the HTML content of the page, called a **template**.

Each Angular application has a main HTML file, named `index.html`, that exists inside the `src` folder and contains the following `<body>` HTML element:

```
<body>
  <app-root></app-root>
</body>
```

The `<app-root>` tag is used to identify the main component of the application and acts as a container to display its HTML content. It instructs Angular to render the template of the main component inside that tag. We will learn how it works in *Chapter 3, Structuring User Interfaces with Components*.

When the Angular CLI builds an Angular application, it parses the `index.html` file and identifies HTML tags inside the `<body>` element. An Angular application is always rendered inside the `<body>` element and comprises a tree of components. When the Angular CLI finds a tag that is not a known HTML element, such as `<app-root>`, it starts searching through the components of the application tree. But how does it know where to start?

Bootstrapping

The startup method of an Angular application is called **bootstrapping**, and it is defined in the `main.ts` file inside the `src` folder:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';
```

```
bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

The main task of the bootstrapping file is to define the component that will be loaded at application startup. It calls the `bootstrapApplication` method, passing `AppComponent` as a parameter to specify the starting component of the application. It also passes the `appConfig` object as a second parameter to specify the configuration that will be used in the application startup. The application configuration is described in the `app.config.ts` file:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/
core';
import { provideRouter } from '@angular/router';

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

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }),
  provideRouter(routes)]
};
```

The `appConfig` object contains a `providers` property to define services provided throughout the Angular application. We will learn more about services in *Chapter 5, Managing Complex Tasks with Services*.

A new Angular CLI application provides routing services by default. Routing is related to in-app navigation between different components using the browser URL. It is activated using the `provideRouter` method, passing a `routes` object, called **route configuration**, as a parameter. The route configuration of the application is defined in the `app.routes.ts` file:

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

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

Our application does not have a route configuration yet, as indicated by the empty `routes` array. We will learn how to set up routing and configure it in *Chapter 9, Navigating through Applications with Routing*.

Template syntax

Now that we have taken a brief overview of our sample application, it's time to start interacting with the source code:

1. Run the following command in a terminal window to start the application if it is not running already:

```
ng serve
```



If you are working with VSCode, it is preferable to use its integrated terminal, which is accessible from the **Terminal | New Terminal** option in the main menu.

2. Open the application with your browser at `http://localhost:4200`, and notice the text below the Angular logo that reads `Hello, my-app`. The word `my-app`, which corresponds to the application name, comes from a variable declared in the TypeScript file of the main component. Open the `app.component.ts` file and locate the `title` variable:

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

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

The `title` variable is a **component property** that is used in the component template.

3. Open the `app.component.html` file and go to line 228:

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

The `title` property is surrounded by double curly braces syntax called **interpolation**, which is part of the Angular template syntax. In a nutshell, interpolation converts the value of the `title` property to text and prints it on the page.

Angular uses specific template syntax to extend and enrich the standard HTML syntax in the application template. We will learn more about the Angular template syntax in *Chapter 3, Structuring User Interfaces with Components*.

4. Change the value of the `title` property in the `AppComponent` class to `World`, save the changes, wait for the application to reload, and examine the output in the browser:



Hello, World

Figure 1.2: Landing page title

Congratulations! You have successfully interacted with the source code of your application.

By now, you should have a basic understanding of how Angular works and what the basic parts of an Angular application are. As a reader, you have had to absorb a lot of information so far. However, you will get a chance to get more acquainted with the components in the upcoming chapters. For now, the focus is to get you up and running, by giving you a powerful tool like the Angular CLI and showing you how only a few steps are needed to display an application on the screen.

Angular tooling

One of the reasons that the Angular framework is popular among developers is the rich ecosystem of available tools. The Angular community has built amazing tools to complete and automate various tasks, such as debugging, inspecting, and authoring Angular applications:

- Angular DevTools
- VSCode Debugger
- VSCode Profiles

We will learn how to use each in the following sections, starting with Angular DevTools.

Angular DevTools

Angular DevTools is a browser extension created and maintained by the Angular team. It allows us to inspect and profile Angular applications directly in the browser. It is currently supported by Google Chrome and Mozilla Firefox and can be downloaded from the following browser stores:

- Google Chrome: <https://chrome.google.com/webstore/detail/angular-developer-tools/ienfalfjdbdpebioblfackekamfmbnh>
- Mozilla Firefox: <https://addons.mozilla.org/firefox/addon/angular-devtools>

To open the extension, open the browser developer tools and select the **Angular** tab. It contains three additional tabs:

- **Components:** Displays the component tree of the Angular application
- **Profiler:** Allows us to profile and inspect the Angular application
- **Injector Tree:** Displays the services provided by the Angular application

In this chapter, we will explore how to use the **Components** tab. We will learn how to use the **Profiler** tab in *Chapter 3, Structuring User Interfaces with Components*, and the **Injector Tree** tab in *Chapter 5, Managing Complex Tasks with Services*.

The **Components** tab allows us to preview the components and directives of an Angular application and interact with them. If we select a component from the tree representation, we can view its properties and metadata:

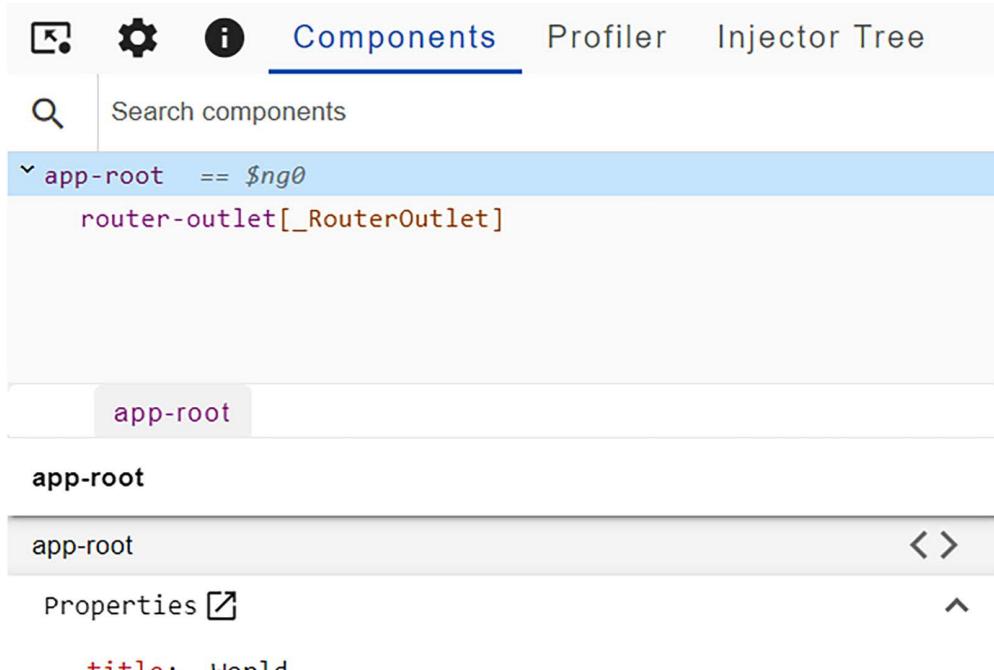
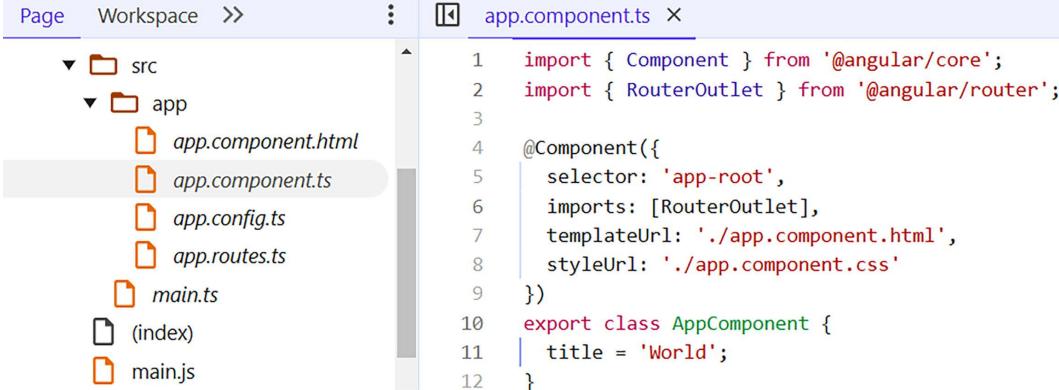


Figure 1.3: Component preview

From the **Components** tab, we can also look up the respective HTML element in the DOM or navigate to the actual source code of the component or directive. Clicking the < > button will take us to the TypeScript file of the current component:



The screenshot shows the Angular CLI workspace interface. On the left, there's a tree view of files under 'src/app': `app.component.html`, `app.component.ts`, `app.config.ts`, `app.routes.ts`, `main.ts`, `(index)`, and `main.js`. The `app.component.ts` file is selected and shown in the main code editor area. The code defines an `AppComponent` with a selector of `'app-root'` and a template URL pointing back to `app.component.html`.

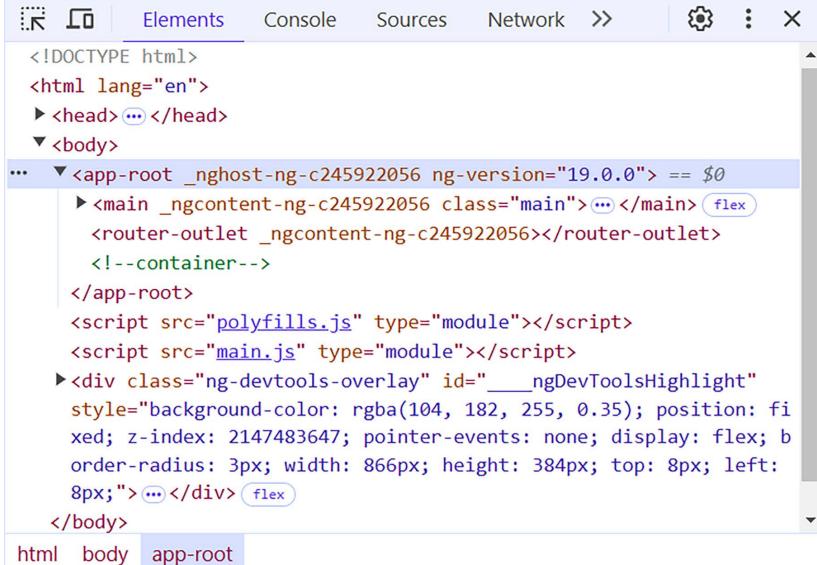
```

1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3
4 @Component({
5   selector: 'app-root',
6   imports: [RouterOutlet],
7   templateUrl: './app.component.html',
8   styleUrls: ['./app.component.css']
9 })
10 export class AppComponent {
11   title = 'World';
12 }

```

Figure 1.4: TypeScript source file

Double-clicking a selector from the tree representation of the **Components** tab will navigate us to the DOM of the main page and highlight the individual HTML element:



The screenshot shows the browser developer tools' Elements tab. It displays the HTML structure of the main page. The `<app-root>` element is highlighted with a blue background, indicating it is the active component. Other elements like `<main>`, `<router-outlet>`, and the `<div ng-devtools-overlay>` are also visible.

Figure 1.5: Main page DOM

Finally, one of the most useful features of the component tree is that we can alter the value of a component property and inspect how the component template behaves:

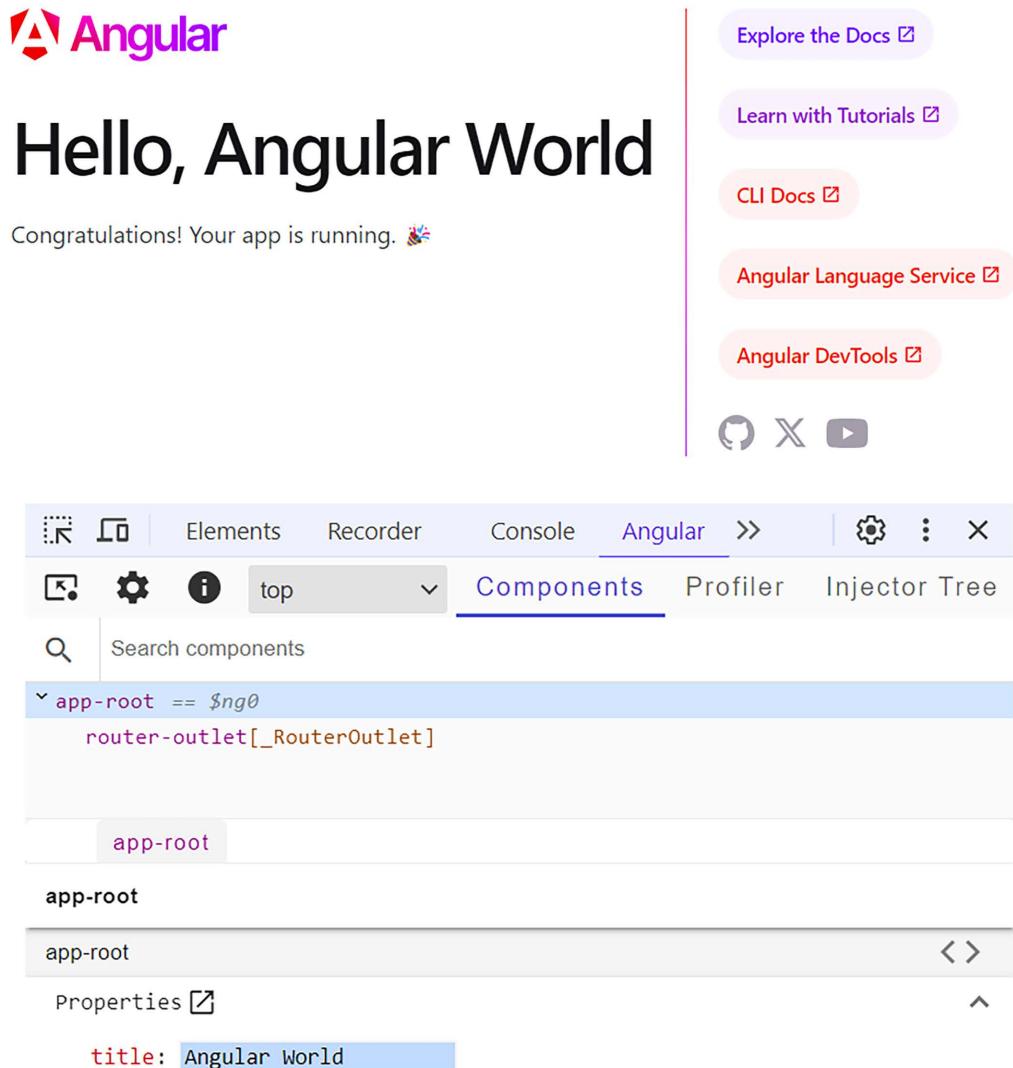


Figure 1.6: Change component state

In the preceding image, you can see that when we changed the value of the `title` property to `Angular World`, the change was also reflected in the component template.

VSCode Debugger

We can debug an Angular application using standard debugging techniques for web applications or the tooling that VSCode provides out of the box.

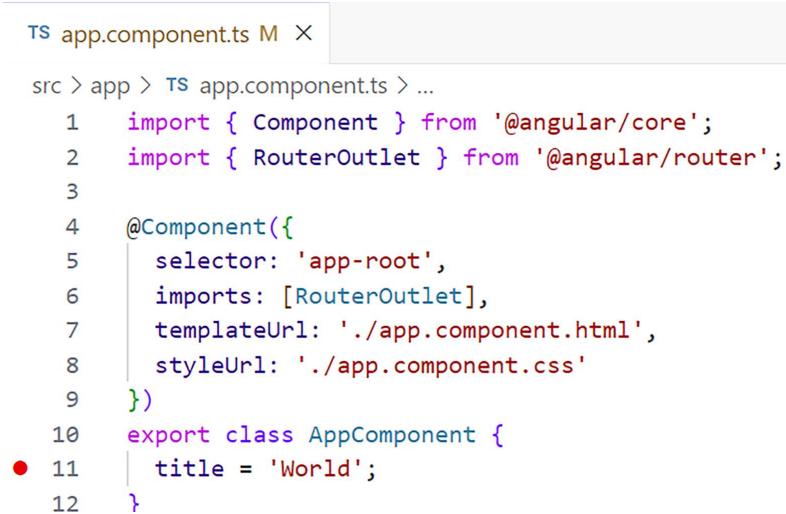
The `console` object is the most commonly used web API for debugging. It is a very fast way to print data and inspect values in the browser console. To inspect the value of an object in an Angular component, we can use the `debug` or `log` method, passing the object we want to inspect as a parameter. However, it is considered an old-fashioned approach, and a codebase with many `console.log` methods is difficult to read. An alternate way is to use **breakpoints** inside the source code using the VSCode debug menu.



VSCode contains a built-in debugging tool that uses breakpoints to debug Angular applications. We can add breakpoints inside the source code from VSCode and inspect the state of an Angular application. When an Angular application runs and hits a breakpoint, it will pause and wait. During that time, we can investigate and inspect several values involved in the current execution context.

Let's see how to add breakpoints to our sample application:

1. Open the `app.component.ts` file and click on the left of line 11 to add a breakpoint. A red dot denotes breakpoints:



```
TS app.component.ts M X
src > app > TS app.component.ts > ...
1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3
4 @Component({
5   selector: 'app-root',
6   imports: [RouterOutlet],
7   templateUrl: './app.component.html',
8   styleUrls: ['./app.component.css']
9 })
10 export class AppComponent {
● 11   title = 'World';
12 }
```

Figure 1.7: Adding a breakpoint

2. Click on the **Run and Debug** button in the left sidebar of VSCode.
3. Click on the play button to start the application using the **ng serve** command:

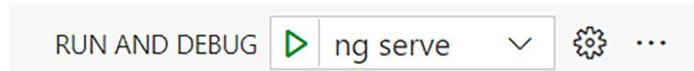
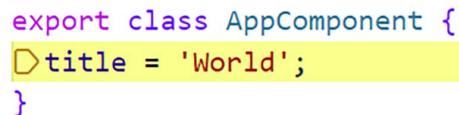


Figure 1.8: Run and debug menu

VSCode will build our application, open the default web browser, and hit the breakpoint inside the editor:



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

Figure 1.9: Hitting a breakpoint

We can now inspect various aspects of our component and use the buttons in the debugger toolbar to control the debugging session.

Another powerful feature of VSCode is **VSCode Profiles**, which help developers customize VSCode according to their development needs.

VSCode Profiles

VSCode Profiles allows us to customize the following aspects of the VSCode editor:

- **Settings:** The configuration settings of VSCode
- **Keyboard shortcuts:** Shortcuts to execute VSCode commands with the keyboard
- **Snippets:** Reusable template code snippets
- **Tasks:** Tasks that automate the execution of scripts and tools directly from VSCode
- **Extensions:** Tools that enable us to add new capabilities in VSCode, such as languages, debuggers, and linters

Profiles can also be shared, which helps us maintain a consistent development setup and workflow across our team. VSCode contains a set of built-in profiles, including one for Angular, that we can further customize according to our development needs. To install the Angular profile:

1. Click the **Manage** button represented by the gear icon at the bottom of the left sidebar in VSCode and select the **Profiles** option.
2. Click on the arrow of the **New Profile** button and select the **From Template | Angular** option.
3. Click the gear button if you want to select a custom icon for your profile.
4. Click the **Create** button to create your profile.

VSCode will automatically apply the new profile after it has been created successfully.

In the following sections, we will explore some of the extensions in the VSCode Angular profile.

Angular Language Service

The **Angular Language Service** extension is developed and maintained by the Angular team and provides code completion, navigation, and error detection inside Angular templates. It enriches VSCode with the following features:

- Code completion
- A go-to definition
- Quick info
- Diagnostic messages

To get a glimpse of its powerful capabilities, let's look at the code completion feature. Suppose we want to display a new property called `description` in the template of the main component. We can set this up by going through the following steps:

1. Define the new property in the `app.component.ts` file:

```
export class AppComponent {  
  title = 'my-app';  
  description = 'Hello World';  
}
```

2. Open the `app.component.html` file and add the property name in the template using Angular interpolation syntax. The Angular Language Service will find it and suggest it for us automatically:

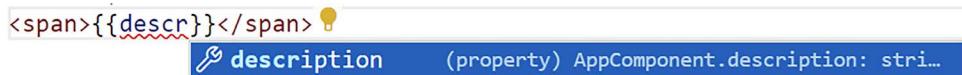


Figure 1.10: Angular Language Service

The `description` property is a *public* property. We can omit the keyword `public` when using public properties and methods. Code completion does not work for private properties and methods. If the property had been declared `private`, then the Angular Language Service and the template would not have been able to recognize it.

You may have noticed that a red line appeared instantly underneath the HTML element as you typed. The Angular Language Service did not recognize the property until you typed it correctly and gave you a proper indication of this lack of recognition. If you hover over the red indication, it displays a complete information message about what went wrong:

```
Property 'descr' does not exist on type 'AppComponent'. ngtsc(2339)  
app.component.ts(1, 23): Error occurs in the template of component AppComponent.  
any  
View Problem (Alt+F8) Quick Fix... (Ctrl+.)
```

Figure 1.11: Error handling in the template

The preceding information message comes from the diagnostic messages feature. The Angular Language Service supports various messages according to the use case. You will encounter more of these messages as you work more with Angular.

Material Icon Theme

VSCode has a built-in set of icons to display different types of files in a project. The Material Icon Theme extension provides additional icons that conform to the Material Design guidelines by Google; a subset of this collection targets Angular-based artifacts:

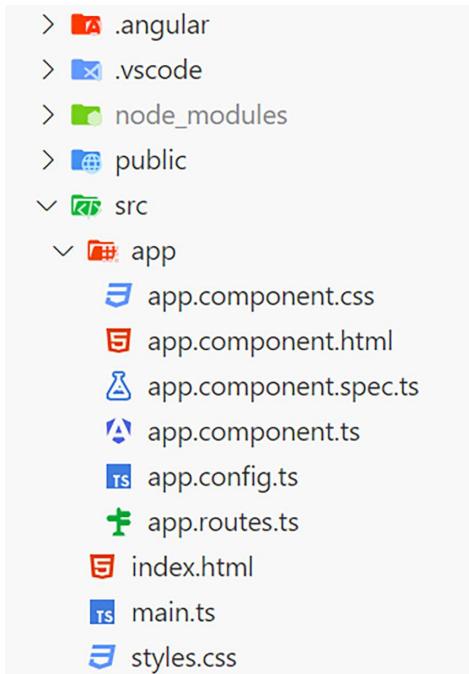


Figure 1.12: Material Icon Theme

Using this extension, you can easily spot the type of Angular files in a project, such as components, and increase developer productivity, especially in large projects with many files.

EditorConfig

VSCode editor settings, such as indentation or spacing, can be set at a user or project level. **EditorConfig** can override these settings using the `.editorconfig` configuration file, which can be found in the root folder of an Angular CLI project:

```
# Editor configuration, see https://editorconfig.org
root = true

[*]
charset = utf-8
```

```
indent_style = space
indent_size = 2
insert_final_newline = true
trim_trailing whitespace = true

[*.ts]
quote_type = single
ij_typescript_use_double_quotes = false

[*.md]
max_line_length = off
trim_trailing whitespace = false
```

You can define unique settings in this file to ensure the consistency of the coding style across your team.

Summary

That's it! Your journey into the world of Angular has just begun. Let's recap the features that you have learned so far. We learned what Angular is, looked over the brief history of the framework, and examined the benefits of using it for web development.

We saw how to set up our development workspace and find the tools to bring TypeScript into the game. We introduced the Angular CLI tool, the Swiss army knife for Angular, which automates specific development tasks. We used some of the most common commands to scaffold our first Angular application. We also examined the structure of our application and learned how to interact with it.

Our first application gave us a basic understanding of how Angular works internally to render our application on a web page. We embarked on our journey, starting with the main HTML file of an Angular application. We saw how Angular parses that file and starts searching the component tree to load the main component. We learned the process of Angular bootstrapping and how it is used to load the application configuration.

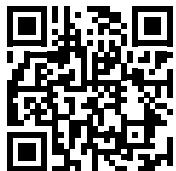
Finally, we met some of the most important Angular tools that could empower you as a software developer. We explored how to use Angular DevTools to inspect Angular applications and VSCode Debugger for debugging. We also examined VSCode Profiles and how it can help us maintain a consistent development environment across our team.

In the next chapter, you will learn some of the basics of the TypeScript language. The chapter will cover what problems can be solved by introducing types and the language itself. TypeScript, as a superset of JavaScript, contains a lot of powerful concepts and marries well with the Angular framework, as you are about to discover.

Join us on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/LearningAngular5e>



2

Introduction to TypeScript

As we learned in the previous chapter, when we built our very first Angular application, the code of an Angular project is written in TypeScript. Writing in TypeScript and leveraging its static typing gives us a remarkable advantage over other scripting languages. This chapter is not a thorough overview of the TypeScript language. Instead, we'll focus on the core elements that will be useful for this book. As we will see very soon, having sound knowledge of these mechanisms is paramount to understanding how dependency injection works in Angular.

In this chapter, we're going to cover the following main topics:

- JavaScript essentials
- What is TypeScript?
- Getting started with TypeScript

We will first refresh our knowledge of JavaScript by revisiting some essential features related to TypeScript, such as functions and classes. We will then investigate the background of TypeScript and the rationale behind its creation. We will also learn how to code and execute TypeScript code. We will emphasize the typing system, which is the main advantage of TypeScript, and learn how to use it to create basic types and interfaces.

Technical requirements

- **GitHub:** <https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition/tree/main/ch02>
- **Node.js:** <https://nodejs.org>
- **Git:** <https://git-scm.com>
- **VSCode:** <https://code.visualstudio.com>

JavaScript essentials

JavaScript is a programming language that contains many features for building web applications. In this section, we will revisit and refresh our knowledge of some of the most basic ones as they are directly correlated with TypeScript and Angular development. TypeScript is a syntactic superset of JavaScript, meaning that it adds features such as types, interfaces, and generics. We will look at the following JavaScript features in more detail:

- Variable declaration
- Function parameters
- Arrow functions
- Optional chaining
- Nullish coalescing
- Classes
- Modules



You can run all the code samples in this section in the following ways:

- Enter the code in a browser console window
- Type the code in a JavaScript file and use Node.js to execute it

If you are comfortable with these features, you can skip directly to the *What is TypeScript?* section.

Variable declaration

Traditionally, JavaScript developers have used the keyword `var` to declare objects, variables, and other artifacts. The reason was that the old semantics of the language only had a function scope where variables were unique within its context:

```
function myFunc() {  
  var x = 0;  
}
```

In the preceding function, no other variable can be declared as `x` inside its body. If you do declare one, then you effectively redefine it. However, there are cases in which scoping is not applied, such as in loops:

```
var x = 20;
```

```
for (var x = 0; x < 10; x++) {  
}
```

In the preceding snippet, the `x` variable outside the loop will not affect the `x` variable inside because they have a different scope. To overcome the scope limitation, JavaScript introduced the `let` keyword:

```
function myFunc() {  
    let x = 0;  
    x = 10;  
}
```

The `let` keyword allows us to change the reference of a variable multiple times in the code.

Another way to define variables in JavaScript is the `const` keyword, which indicates that a variable should never change. As a code base grows, changes may happen by mistake, which can be costly. The `const` keyword can prevent these types of mistakes. Consider the following code snippet:

```
const price = 100;  
price = 50;
```

If we try to execute it, it will throw the following error message:

```
TypeError: Assignment to constant variable.
```

The preceding error will come up only at the top level. You need to be aware of this if you declare objects as constants, like so:

```
const product = { price: 100 };  
product.price = 50;
```

Declaring the `product` variable as a `constant` does not prevent the entire object but rather its reference from being edited. So, the preceding code is valid. If we try to change the reference of the variable, we will get the same type of error as before:

```
const product = { price: 100 };  
product = { price: 50 };
```

It is preferable to use the `const` keyword when we are sure that the properties of an object will not change during its lifetime because it prevents the object from accidentally changing.

When we want to combine variables, we can use the **spread parameter** syntax. A spread parameter uses the ellipsis (...) to expand the values of a variable:

```
const category = 'Computing';
const categories = ['Gaming', 'Multimedia'];
const productCategories = [...categories, category];
```

In the preceding snippet, we combine the `categories` array and the `category` item to create a new array. The `categories` array still contains two items, whereas the new array contains three. The current behavior is called **immutability**, which means not changing a variable but creating a new one that comes from the original.



An object is not immutable if its properties can be changed or its properties are an object whose properties can be changed.

We can also use a spread parameter on objects:

```
const product = {
  name: 'Keyboard',
  price: 75
};
const newProduct = {
  ...product,
  price: 100,
  category: 'Computing'
};
```

In the preceding snippet, we didn't change the original `product` object but created a merge between the two. The value of the `newProduct` object will be:

```
{
  name: 'Keyboard',
  price: 100,
  category: 'Computing'
}
```

The `newProduct` object takes the properties from the `product` object, adds new values on top of it, and replaces the existing ones.

Function parameters

Functions in JavaScript are the processing machines we use to analyze input, digest information, and apply the necessary transformations to data. They use parameters to provide data for transforming the state of our application or returning an output that will be used to shape our application's business logic or user interactivity.

We can declare a function to accept default parameters so that the function assumes a default value when it's not explicitly passed upon execution:

```
function addtoCart(productId, quantity = 1) {  
  const product = {  
    id: productId,  
    qty: quantity  
  };  
}
```

If we do not pass a value for the `quantity` parameter while calling the function, we will get a `product` object with `qty` set to 1.



Default parameters must be defined after all *required* parameters in the function signature.

One significant advantage of JavaScript flexibility when defining functions is accepting an unlimited, non-declared array of parameters called **rest parameters**. Essentially, we can define an additional parameter at the end of the arguments list prefixed by an ellipsis (...):

```
function addProduct(name, ...categories) {  
  const product = {  
    name,  
    categories: categories.join(',')  
  };  
}
```

In the preceding function, we use the `join` method to create a comma-separated string from the `categories` parameter. We pass each parameter separately when calling the function:

```
addProduct('Keyboard', 'Computing', 'Peripherals');
```

Rest parameters are beneficial when we don't know how many arguments will be passed as parameters. The `name` property is also set using another useful feature of the JavaScript language. Instead of setting the property in the `product` object explicitly, we used the property name directly. The following snippet is equivalent to the initial declaration of the `addProduct` function:

```
function addProduct(name, ...categories) {  
    const product = {  
        name: name,  
        categories: categories.join(',')  
    };  
}
```

The shorthand syntax for assigning property values can be used only when the parameter name matches the property name of an object.

Arrow functions

In JavaScript, we can create functions in an alternate way called **arrow functions**. The purpose of an arrow function is to simplify the general function syntax and provide a bulletproof way to handle the function scope, which is traditionally handled by the `this` object. Consider the following example, which calculates a product discount given its price:

```
const discount = (price) => {  
    return (price / 100) * 10;  
};
```

The preceding code does not have a `function` keyword, and the function body is defined by an arrow (`=>`). Arrow functions can be simplified further using the following best practices:

- Omit the parentheses in the function parameters when the signature contains one parameter only.
- Omit the curly braces in the function body and the `return` keyword if the function has only one statement.

The resulting function will look much simpler and easier to read:

```
const discount = price => (price / 100) * 10;
```

Let's explain now how arrow functions are related to scope handling. The value of the `this` object can point to a different context, depending on where we execute a function. When we use it inside a callback, we lose track of the upper context, which usually leads us to use conventions such as assigning its value to an external variable. Consider the following function, which logs a product name using the native `setTimeout` function:

```
function createProduct(name) {  
    this.name = name;  
    this.getName = function() {  
        setTimeout(function() {  
            console.log('Product name is:', this.name);  
        });  
    }  
}
```

Execute the `getName` function using the following snippet and observe the console output:

```
const product = new createProduct('Monitor');  
product.getName();
```

The preceding snippet will not print the `Monitor` product name as expected because our code modifies the scope of the `this` object when evaluating the function inside the `setTimeout` callback. To fix it, convert the `setTimeout` function to use an arrow function instead:

```
setTimeout(() => {  
    console.log('Product name is:', this.name);  
});
```

Our code is now simpler and we can use the function scope safely.

Optional chaining

Optional chaining is a powerful feature that can help us with refactoring and simplifying our code. In a nutshell, it can guide our code to ignore the execution of a statement unless a value has been provided somewhere in that statement. Let's look at optional chaining with an example:

```
const getOrder = () => {  
    return {  
        product: {  
            name: 'Keyboard'  
        }  
    }  
}
```

```
    };
};
```

In the preceding snippet, we define a `getOrder` function that returns the product of a particular order. Next, let's fetch the value of the `product` property, making sure that an order exists before reading it:

```
const order = getOrder();
if (order !== undefined) {
    const product = order.product;
}
```

The previous snippet is a precautionary step in case our object has been modified. If we do not check the object and it has become `undefined`, JavaScript will throw an error. However, we can use optional chaining to improve the previous statement:

```
const order = getOrder();
const product = order?.product;
```

The character `?` after the `order` object ensures that the `product` property will be accessed only if the object has a value. Optional chaining also works in more complicated scenarios, such as:

```
const name = order?.product?.name;
```

In the preceding snippet, we also check if the `product` object has a value before accessing its `name` property.

Nullish coalescing

Nullish coalescing is related to providing a default value when a variable is not set. Consider the following example, which assigns a value to the `quantity` variable only if the `qty` variable exists:

```
const quantity = qty ? qty : 1;
```

The previous statement is called a **ternary operator** and operates like a conditional statement. If the `qty` variable does not have a value, the `quantity` variable will be initialized to the default value of 1. We can rewrite the previous expression using nullish coalescing as:

```
const quantity = qty ?? 1;
```

Nullish coalescing helps us make our code readable and smaller.

Classes

JavaScript classes allow us to structure our application code and create instances of each class. A class can have property members, a constructor, methods, and property accessors. The following code snippet illustrates what a class looks like:

```
class User {  
    firstName = '';  
    lastName = '';  
    #isActive = false;  
  
    constructor(firstName, lastName, isActive = true) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.#isActive = isActive;  
    }  
  
    get fullname() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
  
    get active() {  
        return this.#isActive;  
    }  
}
```

The class statement wraps several elements that we can break down:

- **Member:** The `User` class contains the `firstName`, `lastName`, and `#isActive` members. Class members will only be accessible from within the class itself. Instances of the `User` class will have access only to the public properties `firstName` and `lastName`. The `#isActive` property will not be available because it is private, as denoted by the `#` character in front of the property name.
- **Constructor:** The `constructor` is executed when we create an instance of the class. It is usually used to initialize the class members inside it with the parameters provided in the signature. We can also provide default values for parameters such as the `isActive` parameter.

- **Method:** A method represents a function and may return a value, such as the `getFullname` method, which constructs the full name of a user. It can also be defined as private, similar to class members.
- **Property accessor:** A property accessor is defined by prefixing a method with the `set` keyword to make it writable and the `get` keyword to make it readable, followed by the property name we want to expose. The `active` method is a property accessor that returns the value of the `#isActive` member.

A class can also extend members and functionality of other classes. We can make a class inherit from another by appending the `extends` keyword to the class definition followed by the class we want to inherit:

```
class Customer extends User {  
    taxNumber = '';  
  
    constructor(firstName, lastName) {  
        super(firstName, lastName);  
    }  
}
```

In the preceding snippet, the `Customer` class extends the `User` class, which exposes `firstName` and `lastName` properties. Any instance of the `Customer` class can use those properties by default. We can also override methods from the `User` class by appending a method with the same name. The `constructor` is required to call the `super` method, which points to the `constructor` of the `User` class.

Modules

As our applications scale and grow, there will be a time when we need to organize our code better and make it sustainable and reusable. Modules are a great way to accomplish these tasks, so let's look at how they work and how we can implement them in our application.

In the preceding section, we learned how to work with classes. Having both classes in the same file is not scalable, and maintaining it won't be easy. Imagine how much code you must process to make a simple change in one of the classes. Modules allow us to separate our application code into single files, enforcing the **Single Responsibility Pattern (SRP)**. Each file is a different module concerned with a specific feature or functionality.



A good indication to split a module into multiple files is when the module starts to occupy different domains. For example, a products module cannot contain logic for customers.

Let's refactor the code described in the previous section so that the `User` and `Customer` classes belong to separate modules:

1. Open VSCode and create a new JavaScript file named `user.js`.
2. Enter the contents of the `User` class and add the `export` keyword in the class definition. The `export` keyword makes the module available to other modules and forms the public API of the module.
3. Create a new JavaScript file named `customer.js` and add the contents of the `Customer` class. The `Customer` class cannot recognize the `User` class because they are in different files.
4. Import the `User` class into the `customer.js` file by adding the following statement at the top of the file:

```
import { User } from './user';
```

We use the `import` keyword and the relative path of the module file without the extension to import the `User` class. If a module exports more than one artifact, we place them inside curly braces separated by a comma, such as:

```
import { User, UserPreferences } from './user';
```

Exploring modules concludes our journey of the JavaScript essentials. In the following section, we will learn about TypeScript and how it helps us build web applications.

What is TypeScript?

Transforming small web applications into thick monolithic clients was impossible due to the limitations of earlier JavaScript versions. In a nutshell, large-scale JavaScript applications suffered from serious maintainability and scalability problems as soon as they grew in size and complexity. This issue became more relevant as new libraries and modules required seamless integration into our applications. The lack of proper mechanisms for interoperability led to cumbersome solutions.

To overcome those difficulties, Microsoft built a superset of the JavaScript language that would help build enterprise applications with a lower error footprint using static type checking, better tooling, and code analysis. TypeScript 1.0 was introduced in 2014. It ran ahead of JavaScript, implemented the same features, and provided a stable environment for building large-scale applications. It introduced optional static typing through type annotations, thereby ensuring type checking at compile time and catching errors early in the development process. Its support for declaration files also enabled developers to describe the interface of their modules so that other developers could better integrate them into their code workflow and tooling.



The official TypeScript website can be reached at <https://www.typescriptlang.org>. It contains extensive language documentation and a playground that gives us access to a quick tutorial to get up to speed with the language in no time. It includes some ready-made code examples that cover some of the most common traits of the language.

As a superset of JavaScript, one of the main advantages of embracing TypeScript in your next project is the low entry barrier. If you know JavaScript, you are pretty much all set since all the additional features in TypeScript are optional. You can pick and introduce any of them to achieve your goal. Overall, there is a long list of solid arguments for using TypeScript in your next project, and all apply to Angular.

Here is a short rundown of some of the advantages:

- Annotating your code with types ensures the consistent integration of your different code units and improves code readability and comprehension.
- The built-in type-checker analyzes your code at compile time and helps you prevent errors before executing your code.
- The use of types ensures consistency across your application. Combined with the previous two, the overall code error footprint is minimized in the long run.
- Interfaces ensure the smooth and seamless integration of your libraries in other systems and code bases.
- Language support across different IDEs is amazing, and you can benefit from features such as highlighting code, real-time type checking, and automatic compilation at no cost.
- The syntax is familiar to developers from other OOP-based backgrounds, such as Java, C#, and C++.

In the following section, we will learn how to develop and execute a TypeScript application. In Angular applications, we do not need to execute TypeScript code manually because it is automatically handled by the Angular CLI; however, it is good to know how it works under the hood.

Getting started with TypeScript

The TypeScript language is an npm package that can be installed from the npm registry using the following command:

```
npm install -g typescript
```

In the preceding command, we chose to install TypeScript globally in our system so that we can use it from any path in our development environment. Let's see how we can use TypeScript through a simple example:

1. Open VSCode and select **File | New File...** from the main menu options.
2. Enter **app.ts** in the **New File...** dialog and press *Enter*.

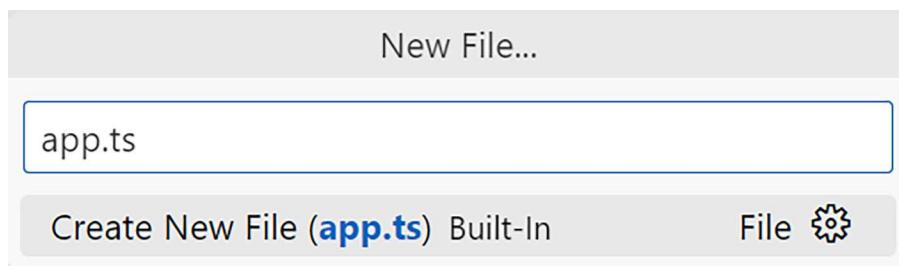


Figure 2.1: New File... dialog

As we have already learned, TypeScript files have a `.ts` extension.

3. Select the path where you want to create the new file. VSCode will then open that file inside the editor.
4. Type the following snippet into the `app.ts` file:

```
const title = 'Hello TypeScript!';
```

Although we have created a TypeScript file, the preceding snippet is valid JavaScript code. Recall that TypeScript is a superset of JavaScript that provides syntactic sugar through its typing system. However, writing plain JavaScript code with TypeScript does not give us any clear benefit.

5. Open a terminal window and run the following command to compile the TypeScript file into JavaScript:

```
tsc app.ts
```

The preceding command initiates a process called **transpilation** performed by the `tsc` executable, a compiler that is at the core of the TypeScript language. We need to compile TypeScript code into JavaScript because browsers do not currently support TypeScript out of the box.



Angular uses a compiler that utilizes the TypeScript compiler under the hood to build Angular applications.

The TypeScript compiler supports extra configuration options that we can pass to the `tsc` executable through the terminal window or a configuration file. The complete list of available compiler options can be found at <https://www.typescriptlang.org/docs/handbook/compiler-options.html>.

6. The transpilation process will create an `app.js` file in the same folder as the TypeScript file. The new file will contain the following code:

```
var title = 'Hello TypeScript!';
```

Since we have not used any specific TypeScript feature yet, the preceding snippet looks almost identical to the original except for the variable declaration.

7. The transpilation process replaced the `const` keyword with the `var` keyword because the TypeScript compiler uses an old JavaScript version by default. We can change that by specifying a target in the `tsc` command:

```
tsc app.ts --target es2022
```

In the preceding command, we specified `es2022`, which represents the most recent version of the JavaScript language at the time of writing. Angular applications that we will build throughout this book also target the same JavaScript version by default.

8. Since we will use the latest JavaScript version in the rest of this chapter, let's define the `target` option using a TypeScript configuration file. Create a file named `tsconfig.json` in the current folder and add the following contents:

```
{  
  "compilerOptions": {  
    "target": "ES2022"  
  }  
}
```

You can find more options for the TypeScript configuration file at <https://www.typescriptlang.org/tsconfig>.

Run the command `tsc` in a terminal window to verify that the output JavaScript file remains unchanged.



When we run the `tsc` command without options, it will compile all TypeScript files in the current folder using the options from the configuration file.

The TypeScript code we have written so far does not use TypeScript-specific features. In the following section, we will learn how to use the typing system, which is the most powerful and essential feature of the TypeScript language.

Types

Working with TypeScript or any other coding language means working with data, and this data can represent different sorts of content, called **types**. Types are used to represent the fact that data can be text, an integer value, or an array of these value types, among others.



Types disappear during transpilation and are not included in the final JavaScript code.

You may have already encountered types in JavaScript since we have always worked implicitly with them. In JavaScript, any given variable could assume (or return, in the case of functions) any value. Sometimes, this leads to errors and exceptions in our code because of type collisions between what our code returned and what we expected to return type-wise. However, statically typing our variables gives our IDE and us a good picture of what kind of data we should find in each code instance. It becomes an invaluable way to help debug our applications at compile time before the code is executed.

String

One of the most widely used primitive types is the `string`, which populates a variable with text:

```
const product: string = 'Keyboard';
```

The type is defined by adding a colon and the type name next to the variable.

Boolean

The `boolean` type defines a variable that can have a value of either `true` or `false`:

```
const isActive: boolean = true;
```

The result of a `boolean` variable represents the fulfillment of a conditional statement.

Number

The `number` type is probably the other most widely used primitive data type, along with `string` and `boolean`:

```
const price: number = 100;
```

We can use the `number` type to define a floating-point number and hexadecimal, decimal, binary, and octal literals.

Array

The `array` type defines a list of items that contain a certain type only. Handling exceptions that arise from errors, such as assigning wrong member types in a list, can now be easily avoided with this type. We can define arrays using the square bracket syntax or the `Array` keyword:

```
const categories: string[] = ['Computing', 'Multimedia'];
const categories: Array<string> = ['Computing', 'Multimedia'];
```



Agreeing with your team on either syntax and sticking with it during application development is advisable.

If we try to add a new item to the `categories` array with a type other than `string`, TypeScript will throw an error, ensuring our typed members remain consistent and that our code is error-free.

any

In all preceding cases, typing is optional because TypeScript is smart enough to infer the data types of variables from their values with a certain level of accuracy.



Letting the typing system infer the types is very important, instead of typing it manually. The type system is never wrong, but the developer can be.

However, if it is not possible, the typing system will automatically assign the dynamic any type to the loosely typed data at the cost of reducing type checking to a bare minimum. Additionally, we can add the any type in our code manually when it is hard to infer the data type from the information we have at any given point. The any type includes all the other existing types, so we can type any data value with it and assign any value to it later:

```
let order: any;
function setOrderNo() {
    order = '0001';
}
```



TypeScript contains another type, similar to the any type, called unknown. A variable of the unknown type can have a value of any type. The main difference is that TypeScript will not let us apply arbitrary operations to unknown values, such as calling a method, unless we perform type checking first.

However, with great power comes great responsibility. If we bypass the convenience of static type checking, we open the door to type errors when piping data through our application. It is up to us to ensure type safety throughout our application.

Custom types

In TypeScript, you can come up with your own type if you need to by using the type keyword in the following way:

```
type Categories = 'computing' | 'multimedia';
```

We can then create a variable of a specific type as follows:

```
const category: Categories = 'computing';
```

The preceding code is perfectly valid as `computing` is one of the allowed values and works as intended. Custom types are an excellent way to add types with a finite number of allowed values.

When we want to create a custom type from an object, we can use the `keyof` operator. The `keyof` operator enables us to iterate over the properties of an object and extract them into a new type:

```
type Category = {  
    computing: string;  
    multimedia: string;  
};  
type CategoryType = keyof Category;
```

In the preceding snippet, the `CategoryType` produced the same result as the `Categories` type. We will learn how we can use the `keyof` operator to iterate over object properties dynamically in *Chapter 4, Enriching Applications Using Pipes and Directives*.

The typing system of TypeScript is mainly used to annotate JavaScript code with types. It improves the developer experience by providing intelliSense and preventing bugs early in development. In the following section, we will learn more about adding type annotations in functions.

Functions

Functions in TypeScript are not that different from regular JavaScript, except that, like everything else in TypeScript, they can be annotated with static types. Thus, they improve the compiler by providing the information it expects in their signature and the data type it aims to return, if any.

The following example showcases how a regular function is annotated in TypeScript:

```
function getProduct(): string {  
    return 'Keyboard';  
}
```

In the preceding snippet, we annotated the returned value of the function by adding the `string` type to the function declaration. We can also add types in function parameters, such as:

```
function getFullname(firstName: string, lastName: string): string {  
    return `${this.firstName} ${this.lastName}`;  
}
```

In the preceding snippet, we annotated the parameters declared in the function signature, which makes sense since the compiler will want to check whether the data provided holds the correct type.



As mentioned in the previous section, the TypeScript compiler is smart enough to infer types when no annotation is provided. In both preceding functions, we could omit the type because the compiler could infer it from the arguments provided and the returned statements.

When a function does not return a type, we can annotate it using the `void` type:

```
function printFullname(firstName: string, lastName: string): void {
    console.log(` ${this.firstName} ${this.lastName}`);
}
```

We have already learned how to use default and rest parameters in JavaScript functions. TypeScript extends functions' capabilities by introducing optional parameters. Parameters are defined as optional by adding the character `?` after the parameter name:

```
function addToCart(productId: number, quantity?: number) {
    const product = {
        id: productId,
        qty: quantity ?? 1
    };
}
```

In the preceding function, we have defined `quantity` as an optional parameter. We have also used the nullish coalescing syntax to set the `qty` property of the `product` object if `quantity` is not passed.

We can invoke the `addToCart` function by passing only the `productId` parameter or both.



Optional parameters should be placed last in a function signature.

We have already learned how JavaScript classes can help us structure our application code. In the following section, we will see how to use them in TypeScript to improve our application further.

Classes

Consider the `User` class that we defined in the `user.js` file:

```
export class User {
    firstName = '';
```

```
lastName = '';
#isActive = false;

constructor(firstName, lastName, isActive = true) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.#isActive = isActive;
}

get fullname() {
    return `${this.firstName} ${this.lastName}`;
}

get active() {
    return this.#isActive;
}
}
```

We will take simple, small steps to add types throughout the class:

1. Convert the file to TypeScript by renaming it `user.ts`.
2. Add the following types to all class properties:

```
firstName: string = '';
lastName: string = '';
private isActive: boolean = false;
```

In the preceding snippet, we also used the `private` modifier to define the `isActive` property as private.

3. Modify the constructor by adding types to parameters:

```
constructor(firstName: string, lastName: string, isActive: boolean =
true) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.isActive = isActive;
}
```



Alternatively, we could omit class properties and have the constructor create them automatically by declaring parameters as `private`:

```
constructor(private firstName: string, private lastName: string, private isActive: boolean = true) {}
```

- Finally, add types in the `active` property accessor and the `getFullscreen` method:

```
getFullscreen(): string {
    return `${this.firstName} ${this.lastName}`;
}

get active(): boolean {
    return this.isActive;
}
```

Converting a JavaScript class into TypeScript and adding types is an important step toward taking advantage of the typing feature in TypeScript.

Another great feature of TypeScript related to classes is the `instanceOf` keyword. It allows us to check the class instance type and provides the correct properties according to the related class. Let's explore it with the `Customer` class defined in the `customer.js` file:

- Convert the file to TypeScript by renaming it `customer.ts`.
- Rewrite the `Customer` class as follows to add types:

```
class Customer extends User {
    taxNumber: number;

    constructor(firstName: string, lastName: string) {
        super(firstName, lastName);
    }
}
```

- Create an object outside of the class that can be of both the `User` and `Customer` type:

```
const account: User | Customer = undefined;
```

4. We can now use the `instanceof` keyword to access different properties of the `account` object according to the underlying class:

```
if (account instanceof Customer) {  
    const taxNo = account.taxNumber;  
} else {  
    const name = account.getFullname();  
}
```

TypeScript is smart enough to understand that the `account` object in the `else` statement does not have a `taxNumber` property because it is of the `User` type. Even if we try to access it by mistake, VSCode will throw an error:

Property 'taxNumber' does not exist on type 'User'. ts(2339)

any

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

Figure 2.2: Property access error

TypeScript classes help us write well-structured code, can be instantiated, contain business logic, and provide static typing in our application. As applications scale and more classes are created, we need to find ways to ensure consistency and rule compliance in our code. As we will learn in the following section, one of the best ways to address the consistency and validation of types is to create **interfaces**.

Interfaces

An interface is a code contract that defines a particular schema. Any artifacts like classes and functions implementing an interface should comply with this schema. Interfaces are beneficial when we want to enforce strict typing on classes generated by factories or when we define function signatures to ensure that a particular typed property is found in the payload.



Interfaces disappear during transpilation and are not included in the final JavaScript code.

In the following snippet, we define an interface for managing products:

```
interface Product {  
    name: string;  
    price: number;  
    getCategories: () => string[];  
}
```



Interfaces are the recommended approach when working with data from a backend API or other source.

An interface can contain properties and methods. In the preceding snippet, the `Product` interface contained the `name` and `price` properties. It also defined the `getCategories` method. A class can use an interface by adding the `implements` keyword and the interface name in the class declaration:

```
class Keyboard implements Product {  
    name: string = 'Keyboard';  
    price: number = 20;  
  
    getCategories(): string[] {  
        return ['Computing', 'Peripherals'];  
    }  
}
```

In the preceding snippet, the `Keyboard` class must implement all members of the `Product` interface; otherwise, TypeScript will throw an error. If we do not want to implement an interface member, we can define it as optional using the `?` character:

```
interface Product {  
    name: string;  
    price: number;  
    getCategories: () => string[];  
    description?: string;  
}
```

We can also use interfaces to change the type of a variable from one type to another, called **type casting**. Type casting is useful when working with dynamic data or when TypeScript cannot infer the type of a variable automatically. In the following code, we instruct TypeScript to treat the product object as a Product type:

```
const product = {
    name: 'Keyboard',
    price: 20
} as Product;
```

However, type casting should be used with caution. In the preceding snippet, we intentionally omitted to add the `getCategories` method, but TypeScript did not throw an error. When we use type casting, we tell TypeScript that a variable *pretends* to be of a specific type.



It is recommended to avoid type casting if possible and define types explicitly.

Interfaces can be combined with **generics** to provide a general code behavior regardless of the data type, as we will learn in the following section.

Generics

Generics are used when we want to use dynamic types in other TypeScript artifacts, such as methods.

Suppose that we want to create a function for saving a `Product` object in the local storage of the browser:

```
function save(data: Product) {
    localStorage.setItem('Product', JSON.stringify(data));
}
```

In the preceding code, we explicitly define the `data` parameter as a `Product`. If we also want to save `Keyboard` objects, we should modify the `save` method as follows:

```
function save(data: Product | Keyboard) {
    localStorage.setItem('Product', JSON.stringify(data));
}
```

However, the preceding approach does not scale well if we would like to add other types in the future. Instead, we can use generics to let the consumer of the `save` method decide upon the data type passed:

```
function save<T>(data: T) {  
  localStorage.setItem('Product', JSON.stringify(data));  
}
```

In the preceding example, the type of `T` is not evaluated until we use the method. We use `T` as a convention to define generics, but you can also use other letters. We can execute the `save` method for a `Product` object as follows:

```
save<Product>({  
  name: 'Microphone',  
  price: 45,  
  getCategories: () => ['Peripherals', 'Multimedia']  
});
```

As you can see, its type varies, depending on how you call it. It also ensures that you are passing the correct type of data. Suppose that the preceding method is called in this way:

```
save<Product>('Microphone');
```

We specify that `T` should be a `Product`, but we insist on passing its value as a string. The compiler clearly states that this is not correct. If we would like to use more generics in our `save` method, we could use different letters, such as:

```
function save<T, P>(data: T, obj: P) {  
  localStorage.setItem('Product', JSON.stringify(data));  
}
```

Generics are often used in collections because they have similar behavior, regardless of the type. They can, however, be used on other constructs, such as methods. The idea is that generics should indicate if you are about to mix types in a way that isn't allowed.

Generics are powerful to use if you have a typical behavior with many different data types. You probably won't be writing custom generics, at least not initially, but it's good to know what is going on.

In the following section, we'll look at some utility types related to interfaces that will help us during Angular development.

Utility types

Utility types are types that help us to derive new types from existing ones.

The `Partial` type is used when we want to create an object from an interface where all its properties are optional. In the following snippet, we use the `Product` interface to declare a trimmed version of a product:

```
const mic: Partial<Product> = {  
    name: 'Microphone',  
    price: 67  
};
```

In the preceding snippet, we can see that the `mic` object does not contain the `getCategories` method. Alternatively, we could use the `Pick` type, which allows us to create an object from a subset of interface properties:

```
type Microphone = Pick<Product, 'name' | 'price'>;  
const microphone: Microphone = {  
    name: 'Microphone',  
    price: 67  
};
```

Some languages, such as C#, have a reserved type when defining a key-value pair object or dictionary, as it is known. In TypeScript, if we want to define such a type, we can use a `Record` type:

```
interface Order {  
    products: Record<string, number>;  
}
```

The preceding snippet defines the product name as a `string` and the quantity as a `number`.

You can find more utility types at <https://www.typescriptlang.org/docs/handbook/utility-types.html>.

Summary

It was a long read, but this introduction to TypeScript was necessary to understand the logic behind many of the most brilliant parts of Angular. It allowed us to introduce the language syntax and explain the rationale behind its success as the syntax of choice for building the Angular framework.

We reviewed the type architecture and how we can create advanced business logic when designing functions with various alternatives for parameterized signatures. We even discovered how to bypass scope-related issues using the powerful arrow functions. We enhanced our knowledge of TypeScript by exploring some of the most common features used in Angular applications.

Probably the most relevant part of this chapter encompassed our overview of classes, methods, properties, and accessors and how we can handle inheritance and better application design through interfaces.

With all this knowledge, we can start learning how to apply it by building Angular applications. In the next chapter, we will learn how to use Angular components to create composable user interfaces to maintain our application code and make it more scalable.

3

Structuring User Interfaces with Components

So far, we have had the opportunity to take a bird's-eye view of the Angular framework. We learned how to create a new Angular application using the Angular CLI and how to interact with an Angular component using template syntax. We also explored TypeScript, which will help us understand how to write Angular code. We have everything we need to explore the further possibilities that Angular brings to the game regarding creating interactive components and how they can communicate with each other.

In this chapter, we will learn about the following concepts:

- Creating our first component
- Interacting with the template
- Component inter-communication
- Encapsulating CSS styling
- Deciding on a change detection strategy
- Introducing the component lifecycle

Technical requirements

This chapter contains various code samples to walk you through Angular components. You can find the related source code in the ch03 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Creating our first component

Components are the basic building blocks of an Angular application. They control different web page parts called **views**, such as a list of products or an order checkout form. They are responsible for the presentational logic of an Angular application, and they are organized in a hierarchical tree of components that can interact with each other:

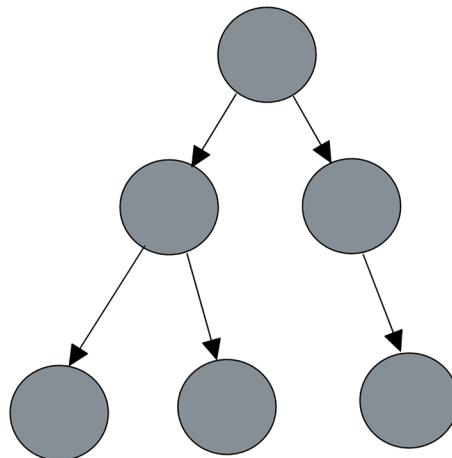


Figure 3.1: Component architecture

The architecture of an Angular application is based on Angular components. Each Angular component can communicate and interact with one or more components in the component tree. As we can see Figure 3.1, a component can simultaneously be a parent of some child components and a child of another parent component.

In this section, we will explore the following topics about Angular components:

- The structure of an Angular component
- Creating components with the Angular CLI

We will start our journey by investigating the internals of Angular components.

The structure of an Angular component

As we learned in *Chapter 1, Building Your First Angular Application*, a typical Angular application contains at least a main component that consists of multiple files. The TypeScript class of the component is defined in the `app.component.ts` file:

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

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

The `@Component` is an **Angular decorator** that defines the properties of the Angular component. An Angular decorator is a method that accepts an object with metadata as a parameter. The metadata is used to configure a TypeScript class as an Angular component using the following properties:

- `selector`: A CSS selector that instructs Angular to load the component in the location that finds the corresponding tag in an HTML template. The Angular CLI adds the `app` prefix by default, but you can customize it using the `--prefix` option when creating the Angular project.
- `imports`: Defines a list of Angular artifacts that the component needs to be loaded correctly, such as other Angular components. The Angular CLI adds the `RouterOutlet` in the main application component by default. The `RouterOutlet` is used when we need routing capabilities in an Angular application. We will learn how to configure routing in *Chapter 9, Navigating through Applications with Routing*.
- `templateUrl`: Defines the path of an external HTML file that contains the HTML template of the component. Alternatively, you can provide the template inline using the `template` property.
- `styleUrl`: Defines the path of an external CSS style sheet file that contains the CSS styles of the component. Alternatively, you can provide the styles inline using the `styles` property.

In applications built with older Angular versions, you may notice that the `imports` property is missing from the `@Component` decorator. This is because such components rely on Angular modules to provide the necessary functionality.



However, starting from Angular v16, the `standalone` property was introduced as an alternative to Angular modules. With Angular v19, **standalone components** are now the default and are enforced throughout the project structure. This shift means that applications created with Angular v19 will utilize the `imports` array in standalone components by default, marking a significant departure from the module-based architecture of earlier versions.

Now that we have explored the structure of an Angular component, we will learn how to use the Angular CLI and create components by ourselves.

Creating components with the Angular CLI

In addition to the main application component, we can create other Angular components that provide specific functionality to the application.



You will need an Angular application to follow along with the rest of the chapter. An option is to create a new Angular application by running the `ng new` command that you learned about in *Chapter 1, Building Your First Angular Application*. Alternatively, you can get the source code from the GitHub repository mentioned in the *Technical requirements* section of the same chapter.

To create a new component in an Angular application, we use the `ng generate` command of the Angular CLI, passing the name of the component as a parameter. Run the following command inside the root folder of the current Angular CLI workspace:

```
ng generate component product-list
```

The preceding command creates a dedicated folder for the component named `product-list` that contains all the necessary files:

- The `product-list.component.css` file, which does not contain any CSS styles yet.
- The `product-list.component.html` file, which contains a paragraph element that displays static text:

```
<p>product-list works!</p>
```

- The `product-list.component.spec.ts` file, which contains a unit test that checks if the component can be created successfully:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ProductListComponent } from './product-list.component';

describe('ProductListComponent', () => {
  let component: ProductListComponent;
  let fixture: ComponentFixture<ProductListComponent>;
```

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [ProductListComponent]
  })
  .compileComponents();

  fixture = TestBed.createComponent(ProductListComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});
});
```

We will learn more about unit testing and its syntax in *Chapter 13, Unit Testing Angular Applications*.

- The `product-list.component.ts` file, which contains the presentational logic of our component:

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

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

In this section, we focused on the TypeScript class of Angular components, but how do they interact with their HTML template?

In the following section, we will learn how to display the HTML template of an Angular component on a page. We will also see how to use the Angular template syntax to interact between the TypeScript class of the component and its HTML template.

Interacting with the template

As we have learned, creating an Angular component using the Angular CLI involves generating a set of accompanying files. One of these files is the component template containing the HTML content displayed on the page. In this section, we will explore how to display and interact with the template through the following topics:

- Loading the component template
- Displaying data from the component class
- Styling the component
- Getting data from the template

We will start our journey in the component template by exploring how we render a component on the web page.

Loading the component template

We learned that Angular uses the `selector` property to load the component in an HTML template. A typical Angular application loads the template of the main component at application startup. The `<app-root>` tag we saw in *Chapter 1, Building Your First Angular Application*, is the selector of the main application component.

To load a component we have created, such as the product list component, we must add its `selector` inside an HTML template. For this scenario, we will load it in the template of the main application component:

1. Open the `app.component.html` file and move the contents of the `<style>` tag in the `app.component.css` file.



It is more maintainable and considered a best practice to have all CSS styles in a separate file.

2. Modify the `app.component.html` file by adding the `<app-product-list>` tag inside the `<div>` tag with the content class:

```
<div class="content">
  <app-product-list></app-product-list>
</div>
```



We can also use self-enclosing tags, similar to `<input>` and `` HTML elements, to add the product list component as `<app-product-list />`.

3. Run the `ng serve` command in a terminal window to start the Angular application. The command will fail, stating the following error:

```
[ERROR] NG8001: 'app-product-list' is not a known element
```

This error is caused because the main application component does not recognize the product list component yet.

4. Open the `app.component.ts` file and import the `ProductListComponent` class:

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.
component';

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

After the application has been built successfully, navigate to `http://localhost:4200` to preview it. The web page displays the static text from the template of the product list component.

In the following sections, we will see how to use the Angular template syntax and interact with the template through the TypeScript class. We will start exploring how to display dynamic data defined in the TypeScript class of the component.

Displaying data from the component class

We have already stumbled upon interpolation to display a property value as text from the component class to the template:

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

Angular converts the `title` component property into text and displays it on the screen.

An alternative way to perform interpolation is to bind the `title` property to the `innerText` property of the `<h1>` HTML element, a method called **property binding**:

```
<h1 [innerText]="title"></h1>
```

In the preceding snippet, we bind to the DOM property of an element and *not* its HTML attribute, as it looks at first sight. The property inside square brackets is called the **target property** and is the property of the DOM element into which we want to bind. The variable on the right is called the **template expression** and corresponds to the `title` property of the component.



When we open a web page, the browser parses the HTML content of the page and converts it into a tree structure, the DOM. Each HTML element of the page is converted into an object called a **node**, which represents part of the DOM. A node defines a set of properties and methods representing the object API. The `innerText` is such a property and is used to set the text inside of an HTML element.

To better understand how the Angular templating mechanism works, we first need to understand how Angular interacts with attributes and properties. It defines HTML attributes to initialize a DOM property and then uses data binding to interact directly with the property.

To set the attribute of an HTML element, we use the `attr.` syntax through property binding followed by the attribute name. For example, to set the `aria-label` accessibility attribute of an HTML element, we would write the following:

```
<p [attr.aria-label]="myText"></p>
```

In the preceding snippet, `myText` is a property of an Angular component. Remember that property binding interacts with the properties of an Angular component. Therefore, if we wanted to set the value of the `innerText` property directly to the HTML, we would write the text value surrounded by single quotes:

```
<h1 [innerText]="'My title'></h1>
```

In this case, the value passed to the `innerText` property is static text, not a component property.

Property binding in the Angular framework binds property values from the component TypeScript class into the template. As we will see next, the **control flow syntax** is suitable for coordinating how those values will be displayed in the template.

Controlling data representation

The new control flow syntax introduced in the latest versions of the Angular framework allows us to manipulate how data will be represented in the component template. It features a set of built-in blocks that add the following capabilities to the Angular template syntax:

- Displaying data conditionally
- Iterating through data
- Switching through templates

In the following sections, we will explore the preceding capabilities, starting with displaying component data based on a conditional statement.

Displaying data conditionally

The `@if` block adds or removes an HTML element in the DOM based on evaluating an expression. If the expression evaluates to true, the element is inserted into the DOM. Otherwise, the element is removed from the DOM. We will illustrate the use of the `@if` block with an example:

1. Run the following command to create an interface for products:

```
ng generate interface product
```

2. Open the `product.ts` file and add the following properties:

```
export interface Product {  
  id: number;  
  title: string;  
}
```

The Product interface defines the structure of a Product object.

3. Open the app.component.css file and move the CSS styles that contain the h1 and p selectors in the product-list.component.css file.
4. Open the product-list.component.ts file and create an empty products array:

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

@Component({
  selector: 'app-product-list',
  imports: [],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
  products: Product[] = [];
}
```

The products array will be used to store a list of Product objects.

5. Open the product-list.component.html file and replace its content with the following snippet:

```
@if (products.length > 0) {
  <h1>Products {{products.length}}</h1>
}
```

The `<h1>` element in the preceding HTML template is rendered on the screen when the products array is not empty. Otherwise, it is removed completely.

6. The `@if` block behaves similarly to a JavaScript `if` statement. Thus, we can add an `@else` section in the component template to execute custom logic when there are not yet any products:

```
@if (products.length > 0) {
  <h1>Products {{products.length}}</h1>
} @else {
  <p>No products found!</p>
}
```

If we had an additional condition that we would like to evaluate, we could use an `@else if` section:



```
@if (products.length > 0) {  
  <h1>Products ({{products.length}})</h1>  
} @else if (products.length === 100) {  
  <span>  
    Click <a>Load More</a> to see more products  
  </span>  
} @else {  
  <p>No products found!</p>  
}
```

7. Run the `ng serve` command to preview the application so far:

No products found!

Figure 3.2: Application output



In applications built with older Angular versions where the control flow syntax is not available, you may notice the `*ngIf` syntax was used to display conditional data:

```
<h1 *ngIf="products.length > 0">  
  Products ({{products.length}})  
</h1>
```

The `*ngIf` is an **Angular directive** with the same behavior as the `@if` block. We will learn how to create custom Angular directives in the following chapter.

However, it is highly recommended to use the `@if` block for the following reasons:

- Makes templates much more readable
- The syntax is closer to JavaScript and is easier to remember
- It is built into the framework and immediately available, which results in smaller bundle sizes

You can find more information about `*ngIf` at <https://angular.dev/guide/directives#adding-or-removing-an-element-with-ngif>.

The application we have built does not display any data because the products array is empty. In the following section, we will learn how to add and display product data on the product list component.

Iterating through data

The @for block allows us to loop through a collection of items and render a template for each, where we can define convenient placeholders to interpolate item data. Each rendered template is scoped to the outer context, where the loop directive is placed so that we can access other bindings. We can think of the @for block as the JavaScript `for` loop but for HTML templates.

We can use the @for block to display the product list in our component as follows:

1. Open the `app.component.css` file and move the CSS styles that contain the `.pill-group`, `.pill`, and `.pill:hover` selectors in the `product-list.component.css` file.
2. Modify the `products` array in the `ProductListComponent` class of the `product-list.component.ts` file so that it contains the following data:

```
export class ProductListComponent {  
  products: Product[] = [  
    { id: 1, title: 'Keyboard' },  
    { id: 2, title: 'Microphone' },  
    { id: 3, title: 'Web camera' },  
    { id: 4, title: 'Tablet' }  
  ];  
}
```

3. Open the `product-list.component.html` file and add the following snippet after the @if block:

```
<ul class="pill-group">  
  @for (product of products; track product.id) {  
    <li class="pill">{{product.title}}</li>  
  }  
</ul>
```

In the preceding code, we use the @for block and turn each item fetched from the `products` array into a `product` variable called the **template input variable**. We reference the template variable in our HTML by binding its `title` property using Angular interpolation syntax.

During the execution of the @for block, data may change, HTML elements may be added, moved, or removed, and the whole list may even be replaced. Angular must synchronize data changes with the DOM tree by connecting the iterated array and its corresponding DOM element. It is a process that can become very slow and expensive and may eventually result in poor performance. For that purpose, Angular relies on the track property, which keeps track of data changes. In our case, the track property defines the property name of the product variable that will be used to keep track of every item in the products array.

4. Run the `ng serve` command to preview the application:

Products (4)

Keyboard

Microphone

Web camera

Tablet

Figure 3.3: Product list

5. The @for block supports adding an @empty section, which is executed when the array of items is empty. We can refactor our code by removing the @else section of the @if block and adding an @empty section as follows:

```
@if (products.length > 0) {  
    <h1>Products ({{products.length}})</h1>  
}  
  
<ul class="pill-group">  
    @for (product of products; track product.id) {  
        <li class="pill">{{product.title}}</li>  
    } @empty {  
        <p>No products found!</p>  
    }  
</ul>
```

The @for block can observe changes in the underlying collection and add, remove, or sort the rendered templates as items are added, removed, or reordered in the collection. It is also possible to keep track of other useful properties as well. We can use the extended version of the @for block using the following syntax:

```
@for (product of products; track product.id; let variable=property) {}
```

The `variable` is a template input variable that we can reference later in our template. The `property` can have the following values:

- `$count`: Indicates the number of items in the array
- `$index`: Indicates the index of the item in the array
- `$first/$last`: Indicates whether the current item is the first or last one in the array
- `$even/$odd`: Indicates whether the index of the item in the array is even or odd



We can use the preceding properties directly or by declaring an alias, as seen in the following example.

In the following snippet, Angular assigns the value of the `$index` property to the `i` input variable. The `i` variable is later used in the template to display each product as a numbered list:

```
@for (product of products; track product.id; let i = $index) {  
  <li class="pill">{{i+1}}. {{product.title}}</li>  
}
```



Use the `$index` property in the `track` variable when unsure of which one you should pick from your object data. Additionally, it is recommended to use it when you don't have any unique property in your object and you are not modifying the order of the list by deleting, adding, or moving elements.

In applications built with older Angular versions, you may notice the following syntax for iterating over collections:



```
<ul class="pill-group">
  <li class="pill" *ngFor="let product of products">
    {{product.title}}
  </li>
</ul>
```

The `*ngFor` is an Angular directive that works similarly to the `@for` block. However, it is highly recommended to use `@for` for the same reasons mentioned about the `@if` block in the previous section.

You can find more information about `*ngFor` at <https://angular.dev/guide/directives#listing-items-with-ngfor>.

The last block of the control flow syntax we will cover is the `@switch` block in the following section.

Switching through templates

The `@switch` block switches between parts of the component template and displays each depending on a defined value.

You can think of `@switch` like the JavaScript `switch` statement. It consists of the following sections:

- `@switch`: Defines the property that we want to check when applying the block
- `@case`: Adds or removes a template from the DOM tree depending on the value of the property defined in the `@switch` block
- `@default`: Adds a template to the DOM tree if the value of the property defined in the `@switch` block does not meet any `@case` statement

We will learn how to use the `@switch` block by displaying a different emoji according to the product title. Open the `product-list.component.html` file and modify the `@for` block so that it includes the following `@switch` block:

```
<ul class="pill-group">
  @for (product of products; track product.id) {
    <li class="pill">
      @switch (product.title) {
        @case ('Keyboard') { ☑ }
```

```
@case ('Microphone') { 🎤 }
@default { 🎵 }
}
{{product.title}}
</li>
} @empty {
<p>No products found!</p>
}
</ul>
```

The `@switch` block evaluates the `title` property of each product. When it finds a match, it activates the appropriate `@case` section. If the value of the `title` property does not match any `@case` section, the `@default` section is activated.

In applications built with older Angular versions, you may notice the following syntax for switching over parts of the template:



```
<div [ngSwitch]="product.title">
  <p *ngSwitchCase="'Keyboard'">⌨️</p>
  <p *ngSwitchCase="'Microphone'">🎤</p>
  <p *ngSwitchDefault>🎵</p>
</div>
```

The `[ngSwitch]` is an Angular directive with the same behavior as the `@switch` block. However, it is highly recommended to use `@switch` for the same reasons mentioned about the `@if` block in the previous section.

You can find more information about `[ngSwitch]` at <https://angular.dev/guide/directives#switching-cases-with-ngswitch>.

The simplicity and improved ergonomics of the control flow syntax have enabled the introduction of the `@defer` block in the Angular framework. The `@defer` block helps to enhance UX and improve application performance by loading parts of the component template asynchronously. We will learn more in *Chapter 15, Optimizing Application Performance*.

In this section, we learned how to leverage the control flow syntax and coordinate how data will be displayed on the component template.



If you want to use this syntax in applications that already use the old directive approach, you can execute the Angular CLI migration described at <https://angular.dev/reference/migrations/control-flow>.

As we will learn in the following section, property binding in the Angular framework applies CSS styles and classes in Angular templates.

Styling the component

Styles in a web application can be applied using either the `class` or `style` attribute, or both, of an HTML element:

```
<p class="star"></p>
<p style="color: greenyellow"></p>
```

The Angular framework provides two types of property binding:

- Class binding
- Style binding

Let's begin our journey on component styling with class binding in the following section.

Class binding

We can apply a single class to an HTML element using the following syntax:

```
<p [class.star]="isLiked"></p>
```

In the preceding snippet, the `star` class will be added to the paragraph element when the `isLiked` expression is true. Otherwise, it will be removed from the element. If we want to apply multiple CSS classes simultaneously, we can use the following syntax:

```
<p [class]="currentClasses"></p>
```

The `currentClasses` variable is a component property. The value of an expression that is used in a class binding can be one of the following:

- A space-delimited string of class names such as '`star active`'.
- An object with keys as the class names and values as boolean conditions for each key. A class is added to the element when the value of the key, with its name, is evaluated to be `true`. Otherwise, the class is removed from the element:

```
currentClasses = {
```

```
  star: true,  
  active: false  
};
```

Instead of styling our elements using CSS classes, we can set styles directly with the style binding.

Style binding

Like the class binding, we can apply single or multiple styles simultaneously using a style binding. A single style can be set to an HTML element using the following syntax:

```
<p [style.color]="'greenyellow'"></p>
```

In the preceding snippet, the paragraph element will have a greenyellow color. Some styles can be expanded further in the binding, such as the width of the paragraph element, which we can define along with the measurement unit:

```
<p [style.width.px]="100"></p>
```

The paragraph element will be 100 pixels long. If we need to toggle multiple styles at once, we can use the object syntax:

```
<p [style]="currentStyles"></p>
```

The currentStyles variable is a component property. The value of an expression that is used in a style binding can be one of the following:

- A string with styles separated by semicolons such as 'color: greenyellow; width: 100px'
- An object where its keys are the names of styles and the values are the actual style values:

```
currentStyles = {  
  color: 'greenyellow',  
  width: '100px'  
};
```

Class and style bindings are powerful features that Angular provides out of the box. Together with the CSS styling configuration that we can define in the @Component decorator, it gives endless opportunities for styling Angular components. An equally compelling feature is the ability to read data from a template into the component class, which we look at next.

Getting data from the template

In the previous section, we learned how to use property binding to display data from the component class. Real-world scenarios usually involve bidirectional data flow through components. To get data from the template back to the component class, we use a technique called **event binding**. We will learn how to use event binding by notifying the component class when a product has been selected from the list:

1. Open the `product-list.component.ts` file and add a `selectedProduct` property:

```
selectedProduct: Product | undefined;
```

2. Open the `product-list.component.html` file and use the interpolation syntax to display the selected product if it exists:

```
@if (selectedProduct) {  
  <p>You selected:  
    <strong>{{selectedProduct.title}}</strong>  
  </p>  
}
```

3. Add a `click` event binding in the `` tag to set the `selectedProduct` to the current product variable of the `@for` block:

```
@for (product of products; track product.id) {  
  <li class="pill" (click)="selectedProduct = product">  
    @switch (product.title) {  
      @case ('Keyboard') { 🖲 }  
      @case ('Microphone') { 🎤 }  
      @default { 💡 }  
    }  
    {{product.title}}  
  </li>  
}
```

4. Running `ng serve` to start the application and click on a product from the list:

Products (4)

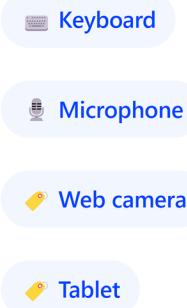


Figure 3.4: Product selection

An event binding listens for DOM events on the target HTML element and responds to those events by interacting with members of the component class. The event inside parentheses is called the **target event** and is the event we are currently listening to. The expression on the right is called the **template statement** and interacts with the component class. Event binding in Angular supports all native DOM events found at <https://developer.mozilla.org/docs/Web/Events>.

The interaction of a component template with its corresponding TypeScript class is summarized in the following diagram:

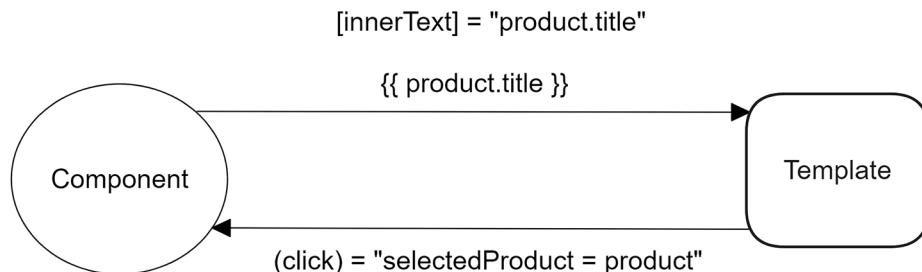


Figure 3.5: Component-template interaction

The same principle we followed for interacting with the component template and class can be used when communicating between components.

Component inter-communication

Angular components expose a public API that allows them to communicate with other components. This API encompasses input properties, which we use to feed the component with data. It also exposes output properties we can bind event listeners to, thereby getting timely information about changes in the component state.

In this section, we will learn how Angular solves the problem of injecting data into and extracting data from components through quick and easy examples.

Passing data using an input binding

The application currently displays the product list and the selected product details in the same component. To learn how to pass data between different components, we will create a new Angular component that will display the details of the selected product. Data representing the specific product details will be dynamically passed from the product list component.

We will start by creating and configuring the component to display product details:

1. Run the following Angular CLI command to create the new Angular component:

```
ng generate component product-detail
```

2. Open the `product-detail.component.ts` file and modify the `import` statements accordingly:

```
import { Component, input } from '@angular/core';
import { Product } from '../product';
```

The `input` function is part of the Signals API and is used when we want to pass data from one component *down* to another component.



We will learn more about the Signals API in *Chapter 7, Tracking Application State with Signals*.

3. Define a `product` property in the `ProductDetailComponent` class that uses the `input` function:

```
export class ProductDetailComponent {
  product = input<Product>();
}
```



In older versions of Angular, we use the `@Input` decorator for passing data between components. You can learn more at <https://angular.dev/guide/components/inputs>.

4. Open the `product-detail.component.html` file and add the following contents:

```
@if (product()) {  
  <p>You selected:  
    <strong>{{product()!.title}}</strong>  
  </p>  
}
```

In the preceding snippet, we use an `@if` block to check if the `product` input property has been set before displaying its title.

5. Open the `product-list.component.ts` file and import the `ProductDetailComponent` class:

```
import { Component } from '@angular/core';  
import { Product } from '../product';  
import { ProductDetailComponent } from '../product-detail/product-  
detail.component';  
  
@Component({  
  selector: 'app-product-list',  
  imports: [ProductDetailComponent],  
  templateUrl: './product-list.component.html',  
  styleUrls: ['./product-list.component.css']  
})
```

6. Finally, replace the last `@if` block in the `product-list.component.html` file with the following snippet:

```
<app-product-detail [product]="selectedProduct"></app-product-  
detail>
```

In the preceding snippet, we use property binding to bind the value of the `selectedProduct` property into the `product` input property of the product detail component. This approach is called **input binding**.

If we run the application and click on a product from the list, we will see that product selection continues to work as expected.

The @if block in the template of the product detail component implies that the product input property is required; otherwise, it does not display its title. Angular does not know if the product list component passes a value for the product input binding during build time. If we want to enforce that rule during compile time, we can define an input property as required accordingly:

```
product = input.required<Product>();
```

According to the previous snippet, if the product list component does not pass a value for the product input property, the Angular compiler will throw the following error:

```
[ERROR] NG8008: Required input 'product' from component  
ProductDetailComponent must be specified.
```

That's it! We have successfully passed data from one component to another. In the following section, we'll learn how to listen for events in a component and respond to them.

Listening for events using an output binding

We learned that input binding is used when we want to pass data between components. This method is applicable in scenarios where we have two components, one that acts as the parent component and the other as the child. What if we want to communicate the other way, from the child component to the parent? How do we notify the parent component about specific actions in the child component?

Consider a scenario where the product detail component should have a button to add the current product to a shopping cart. The shopping cart would be a property of the product list component. How would the product detail component notify the product list component that the button was clicked? Let's see how we would implement this functionality in our application:

1. Open the `product-detail.component.ts` file and import the `output` function from the `@angular/core` npm package:

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

The `output` function is used when we want to create events that will be triggered from one component *up* to another.

2. Define a new component property inside the `ProductDetailComponent` class that uses the `output` function:

```
added = output();
```



In older versions of Angular, we use the `@Output` decorator for triggering events between components. You can learn more at <https://angular.dev/guide/components/outputs>.

3. In the same TypeScript class, create the following method:

```
addToCart() {  
  this.added.emit();  
}
```

The `addToCart` method calls the `emit` method on the `added` output event we created in the previous step. The `emit` method triggers an event and notifies any component currently listening to that event.

4. Now, add a `<button>` element in the component template and bind its `click` event to the `addToCart` method:

```
@if (product()) {  
  <p>You selected:  
    <strong>{{product()!.title}}</strong>  
  </p>  
  <button (click)="addToCart()">Add to cart</button>  
}
```

5. Open the `product-detail.component.css` file and add the following CSS styles that will be applied to the `<button>` element:

```
button {  
  display: flex;  
  align-items: center;  
  --button-accent: var(--bright-blue);  
  background: color-mix(in srgb, var(--button-accent) 65%,  
  transparent);  
  color: white;  
  padding-inline: 0.75rem;  
  padding-block: 0.375rem;  
  border-radius: 0.5rem;  
  border: 0;  
  transition: background 0.3s ease;  
  font-family: var(--inter-font);
```

```
font-size: 0.875rem;
font-style: normal;
font-weight: 500;
line-height: 1.4rem;
letter-spacing: -0.00875rem;
cursor: pointer;
}

button:hover {
  background: color-mix(in srgb, var(--button-accent) 50%, transparent);
}
```

6. We are almost there! Now, we need to wire up the binding in the product list component so that the two components can communicate. Open the `product-list.component.ts` file and create the following method:

```
onAdded() {
  alert(` ${this.selectedProduct?.title} added to the cart! `);
}
```

In the preceding snippet, we use the native `alert` method of the browser to display a dialog to the user.

7. Finally, modify the `<app-product-detail>` tag in the `product-list.component.html` file as follows:

```
<app-product-detail
  [product]="selectedProduct"
  (added)="onAdded()"
></app-product-detail>
```

In the preceding snippet, we use event binding to bind the `onAdded` method into the `added` output property of the product detail component. This approach is called **output binding**.

If we select a product from the list and click on the **Add to cart** button, a dialog box will display a message such as the following:

Web camera added to the cart!

You can see an overview of the component communication mechanism that we have discussed in the following diagram:

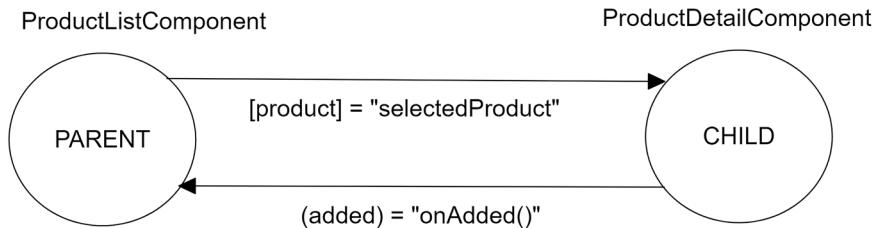


Figure 3.6: Component inter-communication

The output event of the product detail component does nothing more and nothing less than emitting an event to the parent component. However, we can use it to pass arbitrary data through the `emit` method, as we will learn in the following section.

Emitting data through custom events

The `emit` method of an output event can accept any data to pass up to the parent component. It is best to define the data type that can be passed to enforce static type checking.

Currently, the product list component already knows the selected product. Let's assume that the product list component could only realize it after the user clicks on the **Add to cart** button:

1. Open the `product-detail.component.ts` file and use generics to declare the type of data that will be passed into the product list component:

```
added = output<Product>();
```

2. Modify the `addToCart` method so that the `emit` method passes the currently selected product:

```
addToCart() {
  this.added.emit(this.product()!);
}
```

3. Open the `product-list.component.html` file and pass the `$event` variable in the `onAdded` method:

```
<app-product-detail
  [product] = "selectedProduct"
  (added) = "onAdded($event)"
></app-product-detail>
```

The `$event` object is a reserved keyword in Angular that contains the payload data of an event emitter from an output binding, in our case, a `Product` object.

4. Open the `product-list.component.ts` file and change the signature of the `onAdded` method accordingly:

```
onAdded(product: Product) {  
  alert(`${product.title} added to the cart!`);  
}
```

As we saw, output event bindings are a great way to notify a parent component about a change in the component state or send any data.

Besides using the input and output bindings for communicating with components, we can access their properties and methods directly using local **template reference variables**.

Local reference variables in templates

We have seen how to bind data to our templates using interpolation with the double curly braces syntax. Besides this, we often spot named identifiers prefixed by a hash symbol (#) in the elements belonging to our components or even regular HTML elements. These reference identifiers, namely, template reference variables, refer to the components flagged with them in our template views and then access them programmatically. Components can also use them to refer to other elements in the DOM and access their properties.

We have learned how components communicate by listening to emitted events using output binding or passing data through input binding. But what if we could inspect the component in depth, or at least its exposed properties and methods, and access them without going through the input and output bindings? Setting a local reference on the component opens the door to its public API.



The public API of a component consists of all `public` members of the TypeScript class.

We can declare a template reference variable for the product detail component in the `product-list.component.html` file as follows:

```
<app-product-detail  
  #productDetail  
  [product]="${selectedProduct}"
```

```
(added)="onAdded()"  
></app-product-detail>
```

From that moment, we can access the component members directly and even bind them in other locations of the template, such as displaying the product title:

```
<span>{{productDetail.product()!.title}}</span>
```

This way, we do not need to rely on the input and output properties and can manipulate the value of such properties.



The local reference variable approach is particularly useful when using libraries where we cannot control the child components to add input or output binding properties.

We have mainly explained how the component class interacts with its template or other components but have barely been concerned about their styling. We explore that in more detail next.

Encapsulating CSS styling

We can define CSS styling within our components to better encapsulate our code and make it more reusable. In the *Creating our first component* section, we learned how to define CSS styles for a component using an external CSS file through the `styleUrl` property or by defining CSS styles inside the TypeScript component file with the `styles` property.



The usual rules of CSS specificity govern both ways: <https://developer.mozilla.org/docs/Web/CSS/Specificity>.

Thanks to scoped styling, CSS management and specificity become a breeze on browsers that support **shadow DOM**. CSS styles apply to the elements contained in the component, but they do not spread beyond their boundaries.



You can find more detail on shadow DOM at https://developer.mozilla.org/docs/Web/API/Web_components/Using_shadow_DOM.

On top of that, Angular embeds style sheets in the `<head>` element of a web page so that they may affect other elements of our application. We can set up different levels of **view encapsulation** to prevent this from happening.

View encapsulation is how Angular needs to manage CSS scoping within the component. We can change it by setting the `encapsulation` property of the `@Component` decorator in one of the following `ViewEncapsulation` enumeration values:

- **Emulated**: Entails an emulation of native scoping in shadow DOM by sandboxing the CSS rules under a specific selector that points to a component. This option is preferred to ensure that component styles do not leak outside the component and are not affected by other external styles. It is the default behavior in Angular CLI projects.
- **Native**: Uses the native shadow DOM encapsulation mechanism of the renderer that works only on browsers that support shadow DOM.
- **None**: Template or style encapsulation is not provided. The styles are injected as they were added into the `<head>` element of the document. It is the only option if shadow DOM-enabled browsers are not involved.

We will explore the `Emulated` and `None` options due to their extended support using an example:

1. Open the `product-detail.component.html` file and enclose the contents of the `@if` block in a `<div>` element:

```
@if (product()) {  
  <div>  
    <p>You selected:  
      <strong>{{product()!.title}}</strong>  
    </p>  
    <button (click)="addToCart()">Add to cart</button>  
  </div>  
}
```

2. Open the `product-detail.component.css` file and add a CSS style to change the border of a `<div>` element:

```
div {  
  padding-inline: 0.75rem;  
  padding-block: 0.375rem;  
  border: 2px dashed;  
}
```

- Run the application using the `ng serve` command and notice that the product detail component has a dashed border around it when you select a product:

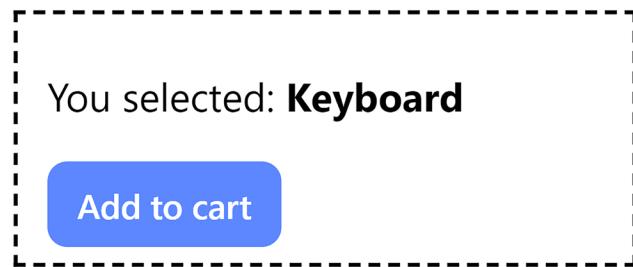
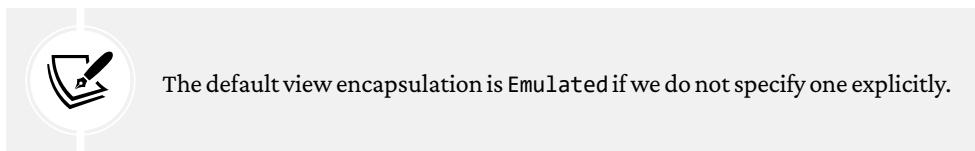


Figure 3.7: Product details

The style did not affect the `<div>` element in the `app.component.html` file because the default encapsulation scopes all CSS styles defined to the specific component.



- Open the `product-detail.component.ts` file and set the component encapsulation to `ViewEncapsulation.None`:

```
import { Component, input, output, ViewEncapsulation } from '@angular/core';
import { Product } from '../product';

@Component({
  selector: 'app-product-detail',
  imports: [],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css'],
  encapsulation: ViewEncapsulation.None
})
```

The application output should look like the following:

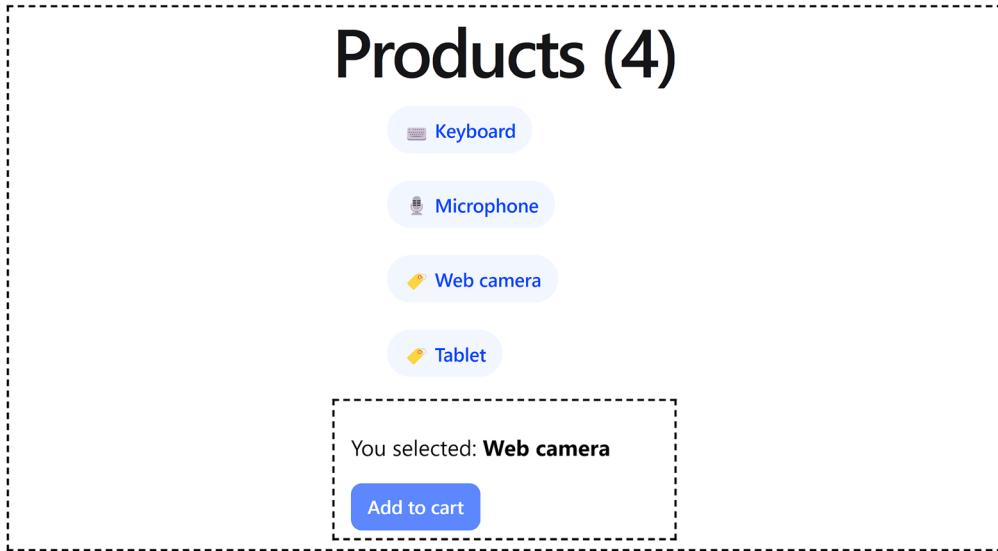


Figure 3.8: No view encapsulation

In the preceding image, the CSS style leaked to the component tree and affected the `<div>` element of the main application component.

View encapsulation can solve many issues when styling our components. However, it should be used cautiously because, as we already learned, CSS styles may leak into parts of the application and produce unwanted effects.

The change detection strategy is another property of the `@Component` decorator that is very powerful. Let's examine this next.

Deciding on a change detection strategy

Change detection is the mechanism that Angular uses internally to detect changes that occur in component properties and reflect these changes to the view. It is triggered on specific events, such as when the user clicks a button, an asynchronous request is completed, or a `setTimeout` and `setInterval` method is executed. Angular uses a process called **monkey patching** to modify such events by overwriting their default behavior using a library called `Zone.js`.

Every component has a change detector that detects whether a change has occurred in its properties by comparing the current value of a property with the previous one. If there are differences, it applies the change to the component template. In the product detail component, when the product input property changes as a result of an event that we mentioned earlier, the change detection mechanism runs for this component and updates the template accordingly.

However, there are cases where this behavior is not desired, such as components that render a large amount of data. In that scenario, the default change detection mechanism is insufficient because it may introduce performance bottlenecks in the application. We could alternatively use the `changeDetection` property of the `@Component` decorator, which dictates the selected strategy the component will follow for change detection.

We will learn how to use a change detection mechanism by profiling our Angular application with Angular DevTools:

1. Open the `product-detail.component.ts` file and create a getter property that returns the current product title:

```
get productTitle() {  
    return this.product()!.title;  
}
```

2. Open the `product-detail.component.html` file and replace the `product.title` expression inside the `` tag with the `productTitle`:

```
@if (product()) {  
    <p>You selected:  
        <strong>{{productTitle}}</strong>  
    </p>  
    <button (click)="addToCart()">Add to cart</button>  
}
```

3. Run the application using the `ng serve` command and preview it at `http://localhost:4200`.
4. Start Angular DevTools, select the **Profiler** tab, and click the **Start recording** button to start profiling the Angular application.

5. Click on the **Keyboard** product from the product list and select the first bar in the bar chart to review change detection:

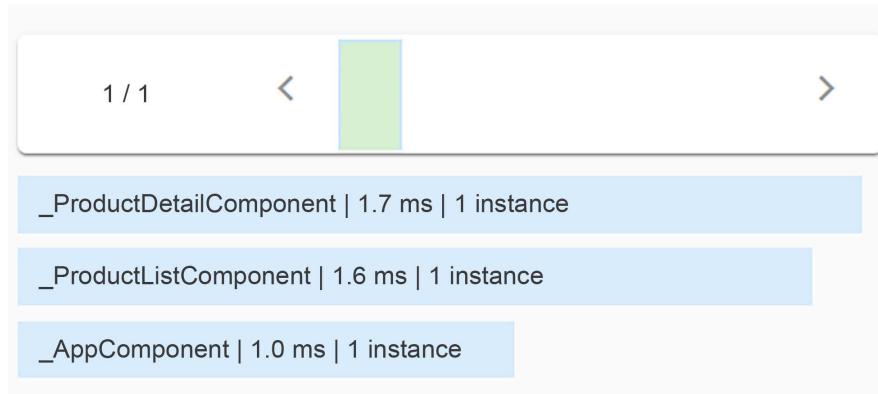


Figure 3.9: Change detection bar chart

In the preceding image, we can see that change detection is triggered for each component in the component tree of the application.

6. Click on the **Add to cart** button and select the second bar in the bar chart:

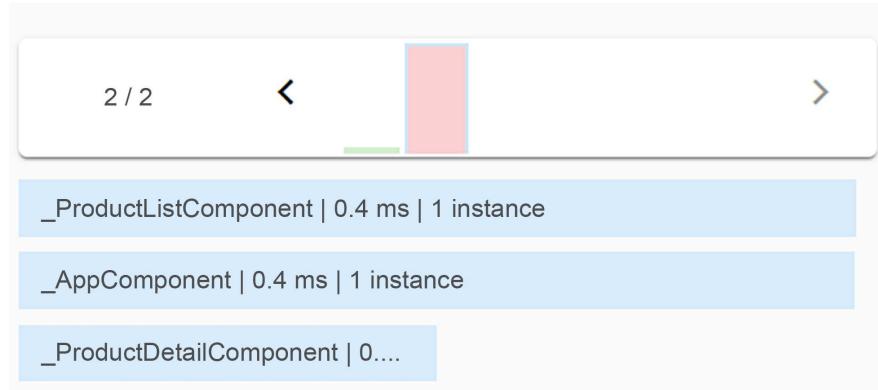


Figure 3.10: Change detection bar chart

Angular executed change detection in the product detail component even though we did not change its properties.

7. Modify the `@Component` decorator of the `product-detail.component.ts` file by setting the `changeDetection` property to `ChangeDetectionStrategy.OnPush`:

```
import { ChangeDetectionStrategy, Component, input, output } from '@angular/core';
import { Product } from '../product';

@Component({
  selector: 'app-product-detail',
  imports: [],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

8. Repeat steps 4 to 6 and observe the output of the second bar in the change detection bar chart:

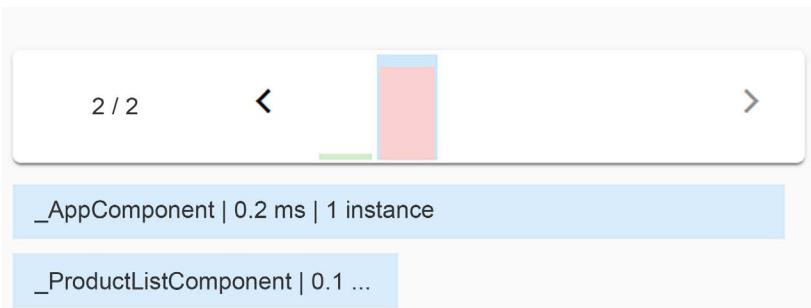


Figure 3.11: Change detection bar chart

Change detection did not run for the product detail component this time.

9. Click on the **Microphone** product from the list and observe the new bar in the bar chart:

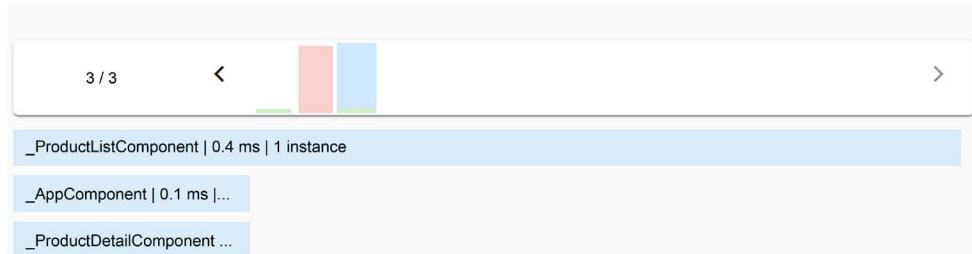


Figure 3.12: Change detection bar chart

Change detection ran this time because we changed the reference of the product input property. If we had just changed a property using the `OnPush` change detection strategy, the change detection mechanism would not have been triggered. You can learn about more change detection scenarios at <https://angular.dev/best-practices/skipping-subtrees>.

The change detection strategy is a mechanism that allows us to modify the way our components detect changes in their data, significantly improving performance in large-scale applications. It concludes our journey of configuring a component, but the Angular framework does not stop there. As we'll learn in the following section, we can hook into specific times in the component lifecycle.

Introducing the component lifecycle

Lifecycle events are hooks that allow us to jump into specific stages in the lifecycle of a component and apply custom logic. They are optional to use but might be valuable if you understand how to use them.

Some hooks are considered best practices, while others help debug and understand what happens in an Angular application. A hook has an interface defining a method we need to implement. The Angular framework ensures the hook is called, provided we have implemented this method in the component.



Defining the interface in the component is not obligatory but is considered a good practice. Angular cares only about whether we have implemented the actual method or not.

The most basic lifecycle hooks of an Angular component are:

- `ngOnInit`: This is called when a component is initialized
- `ngOnDestroy`: This is called when a component is destroyed
- `ngOnChanges`: This is called when values of input binding properties in the component change
- `ngAfterViewInit`: This is called when Angular initializes the view of the current component and its child components

All of these lifecycle hooks are available from the `@angular/core` npm package of the Angular framework.



A full list of all the supported lifecycle hooks is available in the official Angular documentation at <https://angular.dev/guide/components/lifecycle>.

We will explore each one through an example in the following sections. Let's start with the `ngOnInit` hook, which is the most basic lifecycle event of a component.

Performing component initialization

The `ngOnInit` lifecycle hook is a method called during the component initialization. All input bindings and data-bound properties have been set appropriately at this stage, and we can safely use them. Using the component constructor to access them may be tempting, but their values would not have been set at that point. We will learn how to use the `ngOnInit` lifecycle hook through the following example:

1. Open the `product-detail.component.ts` file and add a constructor that logs the value of the `product` property in the browser console:

```
constructor() {  
  console.log('Product:', this.product());  
}
```

2. Import the `OnInit` interface from the `@angular/core` npm package:

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

3. Add the `OnInit` interface to the list of implemented interfaces of the `ProductDetailComponent` class:

```
export class ProductDetailComponent implements OnInit
```

4. Add the following method in the `ProductDetailComponent` class to log the same information as in step 1:

```
ngOnInit(): void {  
  console.log('Product:', this.product());  
}
```

5. Open the `product-list.component.ts` file and set an initial value to the `selectedProduct` property:

```
selectedProduct: Product | undefined = this.products[0];
```

6. Run the application using the `ng serve` command and inspect the output of the browser console:

```
Product: undefined
```

```
Product: ► {id: 1, title: 'Keyboard'}
```

Figure 3.13: Console output

The first message from the constructor contains an `undefined` value, but in the second message, the value of the `product` property is displayed correctly.

Constructors should be relatively empty and devoid of logic other than setting initial variables. Adding business logic inside a constructor makes it challenging to mock it in testing scenarios.

Another good use of the `ngOnInit` hook is when we need to initialize a component with data from an external source, such as an Angular service, as we will learn in *Chapter 5, Managing Complex Tasks with Services*.

The Angular framework provides hooks for all stages of the component lifecycle, from initialization to destruction.

Cleaning up component resources

The interface we use to hook on the destruction event of a component is the `ngOnDestroy` lifecycle hook. We need to import the `OnDestroy` interface and implement the `ngOnDestroy` method to start using it:

```
import { Component, input, OnDestroy, output } from '@angular/core';
import { Product } from '../product';

@Component({
  selector: 'app-product-detail',
  imports: [],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css'
})
export class ProductDetailComponent implements OnDestroy {
```

```

product = input<Product>();
added = output();

addToCart() {
  this.added.emit();
}

ngOnDestroy(): void {

}
}

```

In the preceding snippet, we have added the `OnDestroy` interface and implemented its `ngOnDestroy` method. We can then add any custom logic in the `ngOnDestroy` method to run code when the component is destroyed.

A component is destroyed when it is removed from the DOM tree of a web page due to the following reasons:

- Using the `@if` block from the control flow syntax
- Navigating away from a component using the Angular router, which we will learn about in *Chapter 9, Navigating through Applications with Routing*

We usually perform a cleanup of component resources inside the `ngOnDestroy` method, such as the following:

- Resetting timers and intervals
- Unsubscribing from observable streams, which we will learn about in *Chapter 6, Reactive Patterns in Angular*

An alternative method to the `ngOnDestroy` lifecycle hook is to use a built-in Angular service such as `DestroyRef`:

```

import { Component, DestroyRef, input, output } from '@angular/core';
import { Product } from '../product';

@Component({
  selector: 'app-product-detail',
  imports: [],
  templateUrl: './product-detail.component.html',
})

```

```
    styleUrls: ['./product-detail.component.css'
})
export class ProductDetailComponent {
  product = input<Product>();
  added = output();

  constructor(destroyRef: DestroyRef) {
    destroyRef.onDestroy(() => {
      });
  }

  addToCart() {
    this.added.emit();
  }
}
```

As we will learn in *Chapter 5, Managing Complex Tasks with Services*, using a constructor is one way to inject Angular services into other Angular artifacts. In this case, the `destroyRef` service exposes the `onDestroy` method, which accepts a callback function as a parameter. The callback function will be called when the component is destroyed.

We have already learned how to pass data down to a component using an input binding. The Angular framework provides the `ngOnChanges` lifecycle hook, which we can use to inspect when the value of such a binding has changed.

Detecting input binding changes

The `ngOnChanges` lifecycle hook is called when Angular detects that the value of an input data binding has changed. We will use it in the product detail component to learn how it behaves when we select a different product from the list:

1. Import the `OnChanges` and `SimpleChanges` interfaces in the `product-detail.component.ts` file:

```
import {
  Component,
  input,
  OnChanges,
  output,
```

```
SimpleChanges
} from '@angular/core';
```

2. Modify the definition of the `ProductDetailComponent` class so that it implements the `OnChanges` interface:

```
export class ProductDetailComponent implements OnChanges
```

3. Implement the `ngOnChanges` method that is defined in the `OnChanges` interface. It accepts an object of the `SimpleChanges` type as a parameter that contains one key for each input property that changes. Each key points to another object with the properties `currentValue` and `previousValue`, which denote the new and the old value of the input property, respectively:

```
ngOnChanges(changes: SimpleChanges): void {
  const product = changes['product'];
  const oldValue = product.previousValue;
  const newValue = product.currentValue;
  console.log('Old value', oldValue);
  console.log('New value', newValue);
}
```

The preceding snippet tracks the `product` input property for changes and logs old and new values in the browser console window.

4. To inspect the application, run the `ng serve` command, select a product from the list, and notice the output in the console. You should get something like the following:

```
Old value undefined
New value undefined
Angular is running in development mode.
Old value undefined
New value ▶ {id: 3, title: 'Web camera'}
```

Figure 3.14: Console output

In the preceding image, the first two lines state that the `product` value was changed from `undefined` to `undefined`. It is the actual time when the `product` detail component is initialized, and the `product` property has no value yet. The `OnChanges` lifecycle event is triggered once the value is first set and in all subsequent changes that occur through the binding mechanism.

5. To eliminate the unnecessary log messages, we can check whether it is the first time that the product property is being changed using the isFirstChange method:

```
ngOnChanges(changes: SimpleChanges): void {
  const product = changes['product'];
  if (!product.isFirstChange()) {
    const oldValue = product.previousValue;
    const newValue = product.currentValue;
    console.log('Old value', oldValue);
    console.log('New value', newValue);
  }
}
```

If we refresh the browser, we can see the correct message in the console window.

The ngOnChanges lifecycle hook is a great way to detect when the value of an input property changes. With the advent of the Signals API, we have much better methods to detect and react to these changes, as we will learn in *Chapter 7, Tracking Application State with Signals*. However, for older versions of Angular, the hook is still the preferred solution.

The last lifecycle event of an Angular component we will explore is the ngAfterViewInit hook.

Accessing child components

The ngAfterViewInit lifecycle hook of an Angular component is called when:

- The HTML template of the component has been initialized
- The HTML templates of all child components have been initialized

We can explore how the ngAfterViewInit event works using the product list and product detail components:

1. Open the `product-list.component.ts` file and import the `AfterViewInit` and `viewChild` artifacts from the `@angular/core` npm package:

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

2. Create the following property in the `ProductListComponent` class:

```
productDetail = viewChild(ProductDetailComponent);
```

We have already learned how to query a component class from an HTML template using local reference variables. Alternatively, we can use the `viewChild` function to query a child component from the parent component class.



In older versions of Angular, we use the `@ViewChild` decorator for querying child components. You can learn more at <https://angular.dev/guide/components/queries>.

The `viewChild` function accepts the type of component we want to query as a parameter.

3. Modify the definition of the `ProductListComponent` class so that it implements the `AfterViewInit` interface:

```
export class ProductListComponent implements AfterViewInit
```

4. The `AfterViewInit` interface implements the `ngAfterViewInit` method, which we can use to access the `productDetail` property:

```
ngAfterViewInit(): void {
  console.log(this.productDetail()!.product());
}
```

When we query the `productDetail` property, we get an instance of the `ProductDetailComponent` class. We can then access any member of its public API, such as the `product` property.



Running the preceding code will display an `undefined` value for the `product` property because we do not set an initial value when the product detail component is initialized.

The `ngAfterViewInit` lifecycle event concludes our journey through the lifecycle of Angular components. Component lifecycle hooks are a useful feature of the framework, and you will use them a lot for developing Angular applications.

Summary

In this chapter, we explored Angular components. We saw their structure and how to create them and discussed how to isolate a component's HTML template in an external file to ease its future maintainability. Also, we saw how to do the same with any style sheet we wanted to bind to the component in case we did not want to bundle the component styles inline. We also learned how to use the Angular template syntax and interact with the component template. Similarly, we went through how components communicate bidirectionally using property and event bindings.

We went through the options available in Angular for creating powerful APIs for our components so that we could provide high levels of interoperability between components, configuring their properties by assigning either static values or managed bindings. We also saw how a component could act as a host component for another child component, instantiating the former's custom element in its template and laying the groundwork for larger component trees in our applications. Output parameters give us the layer of interactivity we need by turning our components into event emitters so they can adequately communicate with any parent component that might eventually host them.

Template references paved the way for us to create references in our custom elements, which we can use as accessors to their properties and methods from within the template in a declarative fashion. An overview of the built-in features for handling CSS view encapsulation in Angular gave us additional insights into how we can benefit from shadow DOM's CSS scoping per component. Finally, we learned how important change detection is in an Angular application and how we can customize it to improve its performance further.

We also studied the component lifecycle and learned how to execute custom logic using built-in Angular lifecycle hooks. We still have much more to learn regarding template management in Angular, mostly concerning two concepts you will use in your journey with Angular: directives and pipes, which we will cover in the next chapter.

4

Enriching Applications Using Pipes and Directives

In the previous chapter, we built several components that rendered data on the screen with the help of input and output properties. We'll leverage that knowledge in this chapter to take our components to the next level using Angular **pipes** and **directives**. Pipes allow us to digest and transform the information we bind in our templates. Directives enable more ambitious functionalities, such as manipulating the DOM or altering the appearance and behavior of HTML elements.

In this chapter, we will learn about the following concepts:

- Manipulating data with pipes
- Building pipes
- Building directives

Technical requirements

The chapter contains code samples to walk you through Angular pipes and directives. You can find the related source code in the ch04 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Manipulating data with pipes

Pipes allow us to transform the outcome of our expressions at the view level. They take data as input, transform it into the desired format, and display the output in the template.

The syntax of a pipe consists of the pipe name following the expression we want to transform, separated by a pipe symbol (|):

```
expression | pipe
```

Any parameters are added after the pipe name, and they are separated by colons:

```
expression | pipe:param
```

Pipes can be used with interpolation and property binding in Angular templates and can be chained to each other.

Angular has a wide range of built-in pipe types already baked into it:

- `uppercase/lowercase`: This transforms a string into uppercase or lowercase letters.
- `percent`: This formats a number as a percentage.
- `date`: This formats a date or a string in a particular date format. The default usage of the pipe displays the date according to the local settings of the user's machine. However, we can pass additional formats Angular has already baked in as parameters.
- `currency`: This formats a number as a local currency. We can override local settings and change the currency symbol, passing the currency code as a parameter to the pipe.
- `json`: This takes an object as an input and outputs it in JSON format, replacing single quotes with double quotes. The main usage of the `json` pipe is debugging. It is an excellent way to see what a complex object contains and print it nicely on the screen.
- `keyvalue`: This converts an object into a collection of key-value pairs, where the key of each item represents the object's property and the value is its actual value.
- `slice`: This subtracts a subset (slice) of a collection or string. It accepts as parameters a starting index, where it will begin slicing the input data, and, optionally, an end index. When the end index is specified, the item at that index is not included in the resulting array. If the end index is omitted, it falls back to the last index of the data.



The `slice` pipe transforms immutable data. The transformed list is always a copy of the original data, even when it returns all items.

- `async`: This is used when we manage data handled asynchronously by our component class, and we need to ensure that our views promptly reflect the changes. We will learn more about this pipe later in *Chapter 8, Communicating with Data Services over HTTP*, where we will use it to fetch and display data asynchronously.



You will need the source code of the Angular application we created in *Chapter 3, Structuring User Interfaces with Components*, to follow along with the rest of the chapter.

We will cover the lowercase, currency, and keyvalue pipes in more detail, but we encourage you to explore the rest in the API reference at <https://angular.dev/api>:

1. Open the `product-detail.component.ts` file and import the `CommonModule` class:

```
import { CommonModule } from '@angular/common';
import { Component, input, output } from '@angular/core';
import { Product } from '../product';

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

The `CommonModule` class exports the Angular built-in pipes. An Angular component must import `CommonModule` before using built-in pipes in the component template.

2. Open the `product.ts` file and add the following fields to the `Product` interface that describe additional properties for a product:

```
export interface Product {
  id: number;
  title: string;
  price: number;
  categories: Record<number, string>;
}
```

The `categories` property is an object where the key represents the category ID, and the value represents the category description.

3. Open the `product-list.component.ts` file and modify the `products` array to set values for the new properties:

```
products: Product[] = [
  {
```

```

    id: 1,
    title: 'Keyboard',
    price: 100,
    categories: {
      1: 'Computing',
      2: 'Peripherals'
    }
  },
  {
    id: 2,
    title: 'Microphone',
    price: 35,
    categories: { 3: 'Multimedia' }
  },
  {
    id: 3,
    title: 'Web camera',
    price: 79,
    categories: {
      1: 'Computing',
      3: 'Multimedia'
    }
  },
  {
    id: 4,
    title: 'Tablet',
    price: 500,
    categories: { 4: 'Entertainment' }
  }
];

```

4. Open the `product-detail.component.html` file and add a paragraph element to display the price of the selected product in euros:

```

@if (product()) {
<p>You selected:
  <strong>{{product()!.title}}</strong>
</p>

```

```
<p>{{product()!.price | currency:'EUR'}}</p>
<button (click)="addToCart()">Add to cart</button>
}
```

5. Running `ng serve` to start the application and select the **Microphone** from the product list:

You selected: **Microphone**

€35.00

Add to cart

Figure 4.1: Product details

In the preceding image, the product price is displayed in the currency format.

6. Add the following snippet below the product price to display the product categories:

```
<div class="pill-group">
  @for (cat of product()!.categories | keyvalue; track cat.key) {
    <p class="pill">{{cat.value | lowercase}}</p>
  }
</div>
```

In the preceding snippet, we used the `@for` block to iterate over the `categories` property of the `product` variable. The `categories` property is not iterable because it is a plain object, so, we used the `keyvalue` pipe to convert it into an array that contains `key` and `value` properties. The `key` property represents the category ID, a unique identifier we can use with the `track` variable. The `value` property stores the category description.

Additionally, we used the `lowercase` pipe to convert the category description to lowercase text.

7. Add the following CSS styles to the `product-detail.component.css` file:

```
.pill-group {
  display: flex;
  flex-direction: row;
  align-items: start;
  flex-wrap: wrap;
  gap: 1.25rem;
}
```

```
.pill {
  display: flex;
  align-items: center;
  --pill-accent: var(--gray-900);
  background: color-mix(in srgb, var(--pill-accent) 5%, transparent);
  color: var(--pill-accent);
  padding-inline: 0.75rem;
  padding-block: 0.375rem;
  border-radius: 2.75rem;
  border: 0;
  transition: background 0.3s ease;
  font-family: var(--inter-font);
  font-size: 0.875rem;
  font-style: normal;
  font-weight: 500;
  line-height: 1.4rem;
  letter-spacing: -0.00875rem;
  text-decoration: none;
}
```

8. While running the application, select the **Web camera** product from the list:

You selected: **Web camera**

€79.00

computing

multimedia

Add to cart

Figure 4.2: Product details with categories

Alternative to using the `CommonModule`, we could have imported each pipe class separately from the `@angular/common` npm package:

```
import { CurrencyPipe, KeyValuePipe, LowerCasePipe } from '@angular/common';
```

```
import { Component, input, output } from '@angular/core';
import { Product } from '../product';

@Component({
  selector: 'app-product-detail',
  imports: [KeyValuePipe, CurrencyPipe, LowerCasePipe],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css']
})
```

In the final `product-detail.component.html` file, we use the snippet `product()`! many times to read the value of the `product` property. Alternatively, we could create an alias using the `@let` syntax as follows:

```
@let selectedProduct = product()!;
```

The `@let` keyword is similar to the `let` keyword in JavaScript and is used to declare variables that are available only in the component template. In the preceding snippet, we declare the `selectedProduct` variable, which can be used in the rest of the HTML code as follows:

```
@if (selectedProduct) {
  <p>You selected:
    <strong>{{selectedProduct.title}}</strong>
  </p>
  <p>{{selectedProduct.price | currency:'EUR'}}</p>
  <div class="pill-group">
    @for (cat of selectedProduct.categories | keyvalue; track cat.key) {
      <p class="pill">{{cat.value | lowercase}}</p>
    }
  </div>
  <button (click)="addToCart()">Add to cart</button>
}
```

The `@let` keyword helps us in cases where we want to use complex expressions in templates such as:

- Ternary operators
- Nested object properties
- The `async` pipe

Built-in pipes are sufficient for most use cases, but we must apply complex transformations to our data in other cases. The Angular framework provides a mechanism to create uniquely customized pipes, as we will see in the following section.

Building pipes

We have already seen what pipes are and what their purpose is in the Angular ecosystem. Next, we will dive deeper into how we can build a pipe to provide custom transformations to data bindings. In the following section, we will create a pipe that sorts our list of products by title.

Sorting data using pipes

To create a new pipe, we use the `ng generate` command of the Angular CLI, passing its name as a parameter:

```
ng generate pipe sort
```

The preceding command will generate all necessary files of the `sort` pipe inside the folder where we run the `ng generate` command. The TypeScript class of the pipe is defined in the `sort.pipe.ts` file:

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

@Pipe({
  name: 'sort'
})
export class SortPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

The `@Pipe` is an Angular decorator that defines the name of the Angular pipe.

The TypeScript class of a pipe implements the `transform` method of the `PipeTransform` interface and accepts two parameters:

- `value`: The input data that we want to transform

- args: An optional list of arguments we can provide to the transformation method, each separated by a colon

The Angular CLI helped us by scaffolding an empty `transform` method. We now need to modify it to satisfy our business needs. The pipe will operate on a list of `Product` objects, so we need to make the necessary adjustments to the types provided:

1. Add the following statement to import the `Product` interface:

```
import { Product } from './product';
```

2. Change the type of the `value` parameter to `Product[]` since we want to sort a list of `Product` objects.
3. Change the method type to `Product[]` since the sorted list will only contain `Product` objects, and modify it so that it returns an empty array by default.

The resulting `sort.pipe.ts` file should now look like the following:

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

@Pipe({
  name: 'sort'
})
export class SortPipe implements PipeTransform {

  transform(value: Product[], ...args: unknown[]): Product[] {
    return [];
  }
}
```

We are now ready to implement the sorting algorithm of our method. We will use the native `sort` method, which sorts items alphabetically by default. We will provide a custom comparator function to the `sort` method that overrides the default functionality and performs the sorting logic that we want to achieve:

```
transform(value: Product[], ...args: unknown[]): Product[] {
  if (value) {
    return value.sort((a: Product, b: Product) => {
      if (a.title < b.title) {
```

```
        return -1;
    } else if (b.title < a.title) {
        return 1;
    }
    return 0;
});
}
return [];
}
```

It is worth noting that the `transform` method checks whether there is input data first before proceeding to the sorting process. Otherwise, it returns an empty array. This mitigates cases where the collection is set asynchronously, or the component that consumes the pipe does not set the collection at all.



For more information about the `sort` method, refer to https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/sort.

That's it! We have successfully created our first pipe. We need to call it from our component template to see it in action:

1. Open the `product-list.component.ts` file and import the `SortPipe` class:

```
import { Component } from '@angular/core';
import { Product } from '../product';
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
import { SortPipe } from '../sort.pipe';

@Component({
    selector: 'app-product-list',
    imports: [ProductDetailComponent, SortPipe],
    templateUrl: './product-list.component.html',
    styleUrls: ['./product-list.component.css']
})
```

2. Open the `product-list.component.html` file and add the pipe in the `@for` block:

```
<ul class="pill-group">
  @for (product of products | sort; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      @switch (product.title) {
        @case ('Keyboard') { 🖱 }
        @case ('Microphone') { 🎤 }
        @default { 🌐 }
      }
      {{product.title}}
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

3. If we run the application using the `ng serve` command, we will notice that the product list is now sorted by title alphabetically:

Products (4)

💻 Keyboard

🎤 Microphone

⌚ Tablet

💡 Web camera

Figure 4.3: Product list sorted by title alphabetically

The `sort` pipe can sort product data only by `title`. In the following section, we will learn how to configure the pipe so that it can sort by other product properties as well.

Passing parameters to pipes

As we learned in the *Manipulating data with pipes* section, we can pass additional parameters to a pipe using colons. We use the `args` parameter in the `transform` method of a pipe to get the value of each parameter separated by a colon. We learned that the Angular CLI creates the `args` parameter by default and uses the spread operator to expand its values in the method:

```
transform(value: Product[], ...args: unknown[]): Product[] {
  if (value) {
    return value.sort((a: Product, b: Product) => {
      if (a.title < b.title) {
        return -1;
      } else if (b.title < a.title) {
        return 1;
      }
      return 0;
    });
  }
  return [];
}
```

The `transform` method can currently work only with the `title` property of a product. We could leverage the `args` parameter to make it dynamic and allow the consumer of the pipe to define the property they want to sort data, such as the product price:

1. Remove the spread operator from the `args` parameter because we will pass a single property of a product each time and change its type, as follows:

```
transform(value: Product[], args: keyof Product): Product[] {
  if (value) {
    return value.sort((a: Product, b: Product) => {
      if (a.title < b.title) {
        return -1;
      } else if (b.title < a.title) {
        return 1;
      }
      return 0;
    });
  }
}
```

```
    return [];
}
```

In the preceding method, we use the `keyof` type operator from TypeScript to define that the `args` parameter can be any property of a `Product` object.

2. Replace the `title` property with the `args` parameter inside the `if` statement:

```
if (value) {
  return value.sort((a: Product, b: Product) => {
    if (a[args] < b[args]) {
      return -1;
    } else if (b[args] < a[args]) {
      return 1;
    }
    return 0;
  });
}
```

Notice that in the preceding snippet, we access the `a` and `b` objects using square bracket syntax instead of the dot syntax as before.

3. Modify the `args` parameter in the method signature so that it uses the `title` property by default, if the consumer does not pass any parameter in the pipe:

```
transform(value: Product[], args: keyof Product = 'title')
```

The preceding behavior ensures that the product list component will work without any change to the pipe usage.

4. Run the `ng serve` command and verify that the product list is sorted initially by title.
5. Open the `product-list.component.html` file and pass the `price` property as a pipe parameter:

```
@for (product of products | sort:'price'; track product.id) {
  <li class="pill" (click)="selectedProduct = product">
    @switch (product.title) {
      @case ('Keyboard') {💻}
      @case ('Microphone') {🎙}
      @default {⌚}
    }
}
```

```

    {{product.title}}
</li>
}

```

- Save the file and wait for the application to reload. You should see that the product list is now sorted by price:

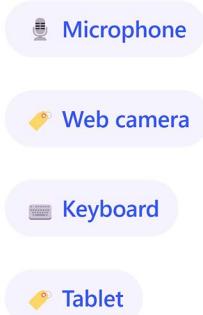


Figure 4.4: Product list sorted by price

The `@Pipe` decorator contains another significant property that we can set, which is directly related to the way that pipes react in the change detection mechanism of the Angular framework.

Change detection with pipes

There are two categories of pipes: `pure` and `impure`. All pipes are considered pure by default unless we set the `pure` property explicitly to `false` in the `@Pipe` decorator:

```

@Pipe({
  name: 'sort',
  pure: false
})

```

Angular executes pure pipes when there is a change to the reference of the input variable. For example, if the `products` array in the `ProductListComponent` class is assigned to a new value, the pipe will correctly reflect that change. However, if we add a new product to the array using the native `Array.push` method, the pipe will not be triggered because the object reference of the array does not change.

Another example is when we have created a pure pipe that operates on a single object. Similarly, if the reference of the value changes, the pipe executes correctly. If a property of the object changes, the pipe cannot detect the change.

A word of caution, however—impure pipes call the `transform` method every time the change detection cycle is triggered. So, this might not be good for performance. Alternatively, you could leave the pure property `unset` and try to cache the value or work with reducers and immutable data to solve this in a better way, like the following:

```
this.products= [
  ...this.products,
  {
    id: 5,
    title: 'Headphones',
    price: 55,
    categories: { 3: 'Multimedia' }
  }
];
```

In the preceding snippet, we used the spread parameter syntax to create a new reference of the `products` array by appending a new item to the reference of the existing array.

Alternatively to a pure pipe, we can use a **computed signal**, which is more effective and ergonomic due to the following reasons:

- We can access the value of the signal in the component class, as opposed to pipes, where their values can be read only in the template
- A computed signal is a simple plain function so we do not need to use a TypeScript class as in pipes

We will learn more about signals in *Chapter 7, Tracking Application State with Signals*.

Creating custom pipes allows us to transform our data in a particular way according to our needs. We must create custom directives if we also want to transform template elements.

Building directives

Angular directives are HTML attributes that extend the behavior or the appearance of a standard HTML element. When we apply a directive to an HTML element or even an Angular component, we can add custom behavior or alter its appearance. There are three types of directives:

- **Components:** Components are directives that contain an associated HTML template.
- **Structural directives:** These add or remove elements from the DOM.

- **Attribute directives:** These modify the appearance of a DOM element or define a custom behavior. We met attribute directives in class and style bindings in the previous chapter.

If a directive has a template attached, then it becomes a component. In other words, components are Angular directives with a view. This rule is handy when deciding whether to create a component or a directive for your needs. If you need a template, create a component; otherwise, make it a directive.

Custom directives allow us to attach advanced behaviors to elements in the DOM or modify their appearance. In the following sections, we will explore how to create attribute directives.

Displaying dynamic data

Attribute directives are commonly used to alter the appearance of an HTML element. We have all probably found ourselves in a situation where we want to add copyrighted information to our applications. Ideally, we want to use this information in various parts of our application, on a dashboard or a contact page. The content of the information should also be dynamic. The year or range of years (it depends on how you want to use it) should update dynamically according to the current date. Our first intention is likely to be to create a component, but what about making it a directive instead? This way, we could attach the directive to any element we want and not bother with a particular template. So, let's begin!

We will use the `ng generate` command of the Angular CLI, passing the name of the directive as a parameter:

```
ng generate directive copyright
```

The preceding command will generate all the necessary files of the copyright directive inside the folder where we run the `ng generate` command. The TypeScript class of the directive is defined in the `copyright.directive.ts` file:

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

@Directive({
  selector: '[appCopyright]'
})
export class CopyrightDirective {

  constructor() { }

}
```

The `@Directive` is an Angular decorator that defines the properties of the Angular directive. It configures a TypeScript class as an Angular directive using the `selector` property. It is a CSS selector that instructs Angular to load the directive in the location that finds the corresponding attribute in an HTML template. The Angular CLI adds the `app` prefix by default, but you can customize it using the `--prefix` option when creating the Angular project.



When we use the selector in an HTML template, we do not add the square brackets.

Let's use the newly created directive to add copyright information to our application:

1. Open the `styles.css` file and add the following CSS styles:

```
.copyright {  
    font-family: "Inter", -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,  
    Helvetica, Arial, sans-serif, "Apple Color Emoji", "Segoe UI Emoji",  
    "Segoe UI Symbol";  
    width: 100%;  
    min-height: 100%;  
    display: flex;  
    justify-content: center;  
    align-items: center;  
    padding: 1rem;  
    box-sizing: inherit;  
    position: relative;  
}
```

In the preceding snippet, we added the CSS styles for our copyright directive in the global CSS stylesheet. Directives do not have an accompanying CSS file that we can use, such as components.

2. Open the `copyright.directive.ts` file and import the `ElementRef` class from the `@angular/core` npm package:

```
import { Directive, ElementRef } from '@angular/core';
```

3. Modify the constructor of the directive as follows:

```
constructor(el: ElementRef) {  
  const currentYear = new Date().getFullYear();  
  const targetEl: HTMLElement = el.nativeElement;  
  targetEl.classList.add('copyright');  
  targetEl.textContent = `Copyright ©${currentYear} All Rights  
Reserved`;  
}
```

In the preceding snippet, we used the `ElementRef` class to access and manipulate the underlying HTML element attached to the directive. The `nativeElement` property contains the actual native HTML element. We also add the `copyright` class using the `add` method of the `classList` property. Finally, we change the text of the element by modifying the `textContent` property.



The `ElementRef` is a built-in Angular service. To use a service in a component or a directive, we need to inject it into the constructor, as we will learn in *Chapter 5, Managing Complex Tasks with Services*.

4. Open the `app.component.ts` file and import the `CopyrightDirective` class:

```
import { Component } from '@angular/core';  
import { RouterOutlet } from '@angular/router';  
import { ProductListComponent } from './product-list/product-list.  
component';  
import { CopyrightDirective } from './copyright.directive';  
  
@Component({  
  selector: 'app-root',  
  imports: [  
    RouterOutlet,  
    ProductListComponent,  
    CopyrightDirective
```

```
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

5. Open the `app.component.html` file and add a `<footer>` element to display copyright information:

```
<main class="main">
  <div class="content">
    <app-product-list></app-product-list>
  </div>
</main>
<footer appCopyright></footer>
<router-outlet />
```

6. Run the application using the `ng serve` command and observe the application output:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Copyright ©2024 All Rights Reserved

Figure 4.5: The application's output

When creating directives, it is important to consider reusable functionality that doesn't necessarily relate to a particular feature. The topic we looked at was copyrighted information, but we could build other functionalities, such as tooltips and collapsible or infinite scrolling features, with relative ease. In the following section, we will build another attribute directive that explores the options available further.

Property binding and responding to events

Attribute directives are also concerned with the behavior of an HTML element. They can extend the functionality of the element and add new features. The Angular framework provides two helpful decorators that we can use in our directives to enhance the functionality of an HTML element:

- `@HostBinding`: This binds a value to the property of the native host element.
- `@HostListener`: This binds to an event of the native host element.



The native host element is the element where our directive takes action.

The native `<input>` HTML element can support different input types, including simple text, radio buttons, and numeric values. When we use the latter, the input adds two arrows inline, up and down, to control its value. It is this feature of the input element that makes it look incomplete. If we type a non-numeric character, the input still renders it.

We will create an attribute directive that rejects non-numeric values entered by the keyboard:

1. Run the following Angular CLI command to create a new directive named `numeric`:

```
ng generate directive numeric
```

2. Open the `numeric.directive.ts` file and import the two decorators that we are going to use:

```
import { Directive, HostBinding, HostListener } from '@angular/core';
```

3. Define a `currentClass` property using the `@HostBinding` decorator that will be bound to the `class` property of the `<input>` element:

```
@HostBinding('class') currentClass = '';
```

4. Define an `onKeyPress` method using the `@HostListener` decorator that will be bound to the `keypress` native event of the `<input>` element:

```
@HostListener('keypress', ['$event']) onKeyPress(event:  
KeyboardEvent) {  
  const keyCode = event.key.charCodeAt(0);  
  if (keyCode > 31 && (keyCode < 48 || keyCode > 57)) {  
    this.currentClass = 'invalid';  
    event.preventDefault();  
  } else {  
    this.currentClass = 'valid';  
  }  
}
```

5. Open the `styles.css` file and add the following CSS styles that will be applied when a component uses the directive:

```
input.valid {  
  border: solid green;  
}  
input.invalid {  
  border: solid red;  
}
```

The `onKeyPress` method contains the logic of how our directive works under the hood.

When the user presses a key inside an `<input>` element, Angular knows to call the `onKeyPress` method because we have registered it with the `@HostListener` decorator. The `@HostListener` decorator accepts the event name and a list of arguments as parameters. In our case, we pass the `keypress` event name and the `$event` argument, respectively. The `$event` is the current object that triggered the event, which is of the `KeyboardEvent` type and contains the keystrokes entered by the user.

Every time the user presses a key, we extract it from the `$event` object, convert it into a Unicode character using the `charCodeAt` method, and check it against a non-numeric code. If the character is non-numeric, we call the `preventDefault` method of the `$event` object to cancel the user action and roll back the `<input>` element to its previous state. At the same time, we set the respective class to `valid` if the key is numeric and `invalid` if it is not.

We can apply the directive in an `<input>` tag as follows:

```
<input appNumeric />
```



We will see a real-world usage of the directive in *Chapter 10, Collecting User Data with Forms*. In the meantime, if you want to try it yourself, remember to import the `NumericDirective` class in your component before using it.

Summary

Now that we have reached this point, it is fair to say that you have met almost every Angular artifact for building Angular components, which are indeed the wheels and the engine of all Angular applications. In the forthcoming chapters, we will see how we can design our application architecture better, manage dependency injection throughout our component tree, consume data services, and leverage the new Angular router to show and hide components when required.

Now, get ready to take on new challenges—in the next chapter, we will discover how to use data services to manage complex tasks in our components.

5

Managing Complex Tasks with Services

We have reached a point in our journey where we can successfully develop more complex applications by nesting components within other components in a sort of component tree. However, bundling all our business logic into a single component is not the way to go. Our application might become unmaintainable very soon as it develops.

In this chapter, we'll investigate the advantages that Angular's dependency management mechanism can bring to the table to overcome such problems. We will learn how to use the **Angular Dependency Injection (DI)** mechanism to declare and consume our dependencies across the application with minimum effort and optimal results. By the end of this chapter, you will be able to create an Angular application that is correctly structured to enforce the **Separation of Concerns (SoC)** pattern using services.

We will cover the following concepts relating to Angular services:

- Introducing Angular DI
- Creating our first Angular service
- Providing dependencies across the application
- Injecting services in the component tree
- Overriding providers in the injector hierarchy

Technical requirements

The chapter contains various code samples to walk you through the concept of Angular services. You can find the related source code in the ch05 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Introducing Angular DI

DI is an application design pattern we also come across in other languages, such as C# and Java. As our applications grow and evolve, each code entity will internally require instances of other objects, better known as **dependencies**. Passing such dependencies to the consumer code entity is known as an **injection**, and it also entails the participation of another code entity called an **injector**. An injector is responsible for instantiating and bootstrapping the required dependencies to be ready for use when injected into a consumer. The consumer knows nothing about how to instantiate its dependencies and is only aware of the interface they implement to use them.

Angular includes a top-notch DI mechanism to expose required dependencies to any Angular artifact of an Angular application. Before delving deeper into this subject, let's look at the problem that DI in Angular is trying to address.

In *Chapter 3, Structuring User Interfaces with Components*, we learned how to display a list of objects using the @for block. We used a static list of Product objects that were declared in the `product-list.component.ts` file, as shown here:

```
products: Product[] = [
  {
    id: 1,
    title: 'Keyboard',
    price: 100,
    categories: {
      1: 'Computing',
      2: 'Peripherals'
    }
  },
  {
    id: 2,
    title: 'Microphone',
    price: 35,
    categories: { 3: 'Multimedia' }
```

```
},
{
  id: 3,
  title: 'Web camera',
  price: 79,
  categories: {
    1: 'Computing',
    3: 'Multimedia'
  }
},
{
  id: 4,
  title: 'Tablet',
  price: 500,
  categories: { 4: 'Entertainment' }
}
];
};
```

This previous approach has two main drawbacks:

- In real-world applications, we rarely work with static data. It usually comes from a back-end API or some other external source.
- The product list is tightly coupled with the component. Angular components are responsible for the presentation logic and should not be concerned with how to get data. They only need to display it in the HTML template. Thus, they should delegate business logic to services to handle such tasks.

In the following section, we'll learn how to avoid these obstacles using Angular services.



You will need the source code of the Angular application we created in *Chapter 4, Enriching Applications Using Pipes and Directives*, to follow along with the rest of the chapter.

We will create an Angular service that will return the product list. Thus, we will effectively delegate business logic tasks away from the component. Remember: *the component should only be concerned with presentation logic*.

Creating our first Angular service

To create a new Angular service, we use the `ng generate` command of the Angular CLI while passing the name of the service as a parameter:

```
ng generate service products
```

Running the preceding command will create the `products` service, which consists of the `products.service.ts` file and its accompanying unit test file, `products.service.spec.ts`.

We usually name a service after the functionality that it represents. Every service has a business context or domain within which it operates. When it starts to cross boundaries between different contexts, this is an indication that you should break it into different services. A `products` service should be concerned with products. Similarly, orders should be managed by a separate `orders` service.

An Angular service is a TypeScript class marked with the `@Injectable` decorator. The decorator identifies the class as an Angular service that can be injected into other Angular artifacts such as components, directives, or even other services. It accepts an object as a parameter with a single property named `providedIn`, which defines which injector provides the service.

An Angular service, by default, is registered with an injector – the root injector of the Angular application, as defined in the `products.service.ts` file:

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

@Injectable({
  providedIn: 'root'
})
export class ProductsService {

  constructor() { }
}
```

Our service does not contain any implementation. Let's add some logic so that our component can use it:

1. Add the following statement to import the `Product` interface:

```
import { Product } from './product';
```

2. Create the following method in the ProductsService class:

```
getProducts(): Product[] {  
    return [  
        {  
            id: 1,  
            title: 'Keyboard',  
            price: 100,  
            categories: {  
                1: 'Computing',  
                2: 'Peripherals'  
            }  
        },  
        {  
            id: 2,  
            title: 'Microphone',  
            price: 35,  
            categories: { 3: 'Multimedia' }  
        },  
        {  
            id: 3,  
            title: 'Web camera',  
            price: 79,  
            categories: {  
                1: 'Computing',  
                3: 'Multimedia'  
            }  
        },  
        {  
            id: 4,  
            title: 'Tablet',  
            price: 500,  
            categories: { 4: 'Entertainment' }  
        }  
    ];  
}
```

In the following sections, we will learn how to use the service in our application.

Injecting services in the constructor

The most common way to use a service in an Angular component is through its constructor:

1. Open the `product-list.component.ts` file and modify the `products` property so that it is initialized to an empty array:

```
products: Product[] = [];
```

2. Add the following statement to import the `ProductsService` class:

```
import { ProductsService } from '../products.service';
```

3. Create a component property called `productService` and give it a type of `ProductsService`:

```
private productService: ProductsService;
```

4. Instantiate the property using the `new` keyword in the component's constructor:

```
constructor() {
  this.productService = new ProductsService();
}
```

5. Import the `OnInit` interface from the `@angular/core` npm package:

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

6. Add the `OnInit` interface to the list of implemented interfaces of the `ProductListComponent` class:

```
export class ProductListComponent implements OnInit
```

7. Add the following `ngOnInit` method that calls the `getProducts` method of the `productService` property and assigns the returned value to the `products` property:

```
ngOnInit(): void {
  this.products = this.productService.getProducts();
}
```

Run the application using the `ng serve` command to verify that the list of products is still shown correctly on the page:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Copyright ©2024 All Rights Reserved

Figure 5.1: Product list

Awesome! We have successfully wired up our component with the service, and our application looks great. Well, this seems to be the case, but it's actually not. There are some problems with the actual implementation. If the `ProductsService` class must change, maybe to accommodate another dependency, `ProductListComponent` should also change the implementation of its constructor. Thus, it is evident that the product list component is tightly coupled to the implementation of `ProductsService`. It prevents us from altering, overriding, or neatly testing the service if required. It also entails that a new `ProductsService` object is created every time we render a product list component, which might not be desired in specific scenarios, such as when we expect to use an actual singleton service.

DI systems try to solve these issues by proposing several patterns, and the **constructor injection** pattern is the one enforced by Angular. We could remove the `productService` component property and inject the service directly into the constructor. The resulting `ProductListComponent` class would be the following:

```
export class ProductListComponent implements OnInit {  
  products: Product[] = [];  
  selectedProduct: Product | undefined;
```

```

constructor(private productService: ProductsService) {}

onAdded() {
  alert(`${this.selectedProduct?.title} added to the cart!`);
}

ngOnInit(): void {
  this.products = this.productService.getProducts();
}
}

```



Consider declaring injected services as `readonly` to provide more stable code and prevent re-assignment of the service. In the preceding snippet, the `constructor` could be re-written as `constructor(private readonly productService: ProductsService) {}.`

The component does not need to know how to instantiate the service. On the other hand, it expects such a dependency to be available before it is instantiated so that it can be injected through its `constructor`. This approach is easier to test as it allows us to override it or mock it up.

However, using a `constructor` is not the only way to inject services in an Angular application, as we will learn in the following section.

The `inject` keyword

The Angular framework contains a built-in `inject` method that we can use to inject services without using the `constructor`. There are some cases where we would like to use the `inject` method:

- The `constructor` contains many injected services, making our code unreadable.
- Constructors cannot be used when working with pure functions in the Angular router or the HTTP client, as we will learn in the following chapters.

Let's see how we could refactor the product list component to use the `inject` method:

1. Open the `product-list.component.ts` file and import the `inject` method from the `@angular/core` npm package:

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

2. Declare the following property in the `ProductListComponent` class:

```
private productService = inject(ProductsService);
```

3. Remove the constructor from the `ProductListComponent` class.

The application should still work as expected if we run the `ng serve` command. The product list should be displayed as in the preceding section.

We will explore additional use cases for the `inject` method in *Chapter 8, Communicating with Data Services over HTTP*, and *Chapter 9, Navigating through Applications with Routing*.

Compared to the constructor approach, the `inject` method provides more accurate types, enforcing strongly typed Angular applications.

The Angular CLI provides a schematic that we can run to migrate to the new `inject` method. You can find more detail on how to run the schematic at <https://angular.dev/reference/migrations/inject-function>.



In this book, we use both the `inject` method and constructor approach, according to the execution context of the application code.

As we learned, when we create a new Angular service, the Angular CLI registers this service with the root injector of the application by default. In the following section, we'll learn about the internals of the DI mechanism and how the root injector works.

Providing dependencies across the application

The Angular framework offers a DI mechanism to provide dependencies in Angular artifacts such as components, directives, pipes, and services. The Angular DI is based on an injector hierarchy where at the top there is the root injector of an Angular application.

Injectors in Angular can examine the dependencies in the constructor of an Angular artifact and return an instance of the type represented by each dependency, so that we can use it straight away in the implementation of our Angular class. The injector maintains a list of all dependencies that an Angular application needs. When a component or other artifact wants to use a dependency, the injector first checks to see if it has already created an instance of this dependency. If not, it creates a new one, returns it to the component, and keeps a copy for further use. The next time the same dependency is requested, it returns the copy previously created. But how does the injector know which dependencies an Angular application needs?

When we create an Angular service, we use the `providedIn` property of the `@Injectable` decorator to define how it is provided to the application. That is, we create a **provider** for this service. A provider is a *recipe* containing guidelines on creating a specific service. During application start-up, the framework is responsible for configuring the injector with providers of services so that it knows how to create one upon request. An Angular service is configured with the root injector by default when created with the CLI. The root injector creates singleton services that are globally available through the application.

In *Chapter 1, Building Your First Angular Application*, we learned that the application configuration object defined in the `app.config.ts` file has a `providers` property where we can register application services. We could remove the `providedIn` property from the `@Injectable` decorator of the `products.service.ts` file and add it in that array directly. Registering a service in this way is the same as configuring the service with `providedIn: 'root'`. The main difference between them is that the `providedIn` syntax is **tree shakable**.



Tree shaking is the process of finding dependencies that are not used in an application and removing them from the final bundle. In the context of Angular, the Angular compiler can detect and delete Angular services that are not used, resulting in a smaller bundle.

When you provide a service through the application configuration object, the Angular compiler cannot say if the service is used somewhere in the application. So, it includes the service in the final bundle *a priori*. Thus, using the `@Injectable` decorator over the `providers` array of the application configuration is preferable.



You should always register singleton services with the root injector.

The root injector is not the only injector in an Angular application. Components have their injectors, too. Angular injectors are also hierarchical. Whenever an Angular component defines a token in its constructor, the injector searches for a type that matches that token in the pool of registered providers. If no match is found, it delegates the search to the parent component's provider and keeps bubbling the component injector tree until it reaches the root injector. If no match is found, Angular throws an exception.

Let's explore the injector hierarchy of the product list component using Angular DevTools:

1. Run the application using the `ng serve` command and preview it at `http://localhost:4200`.
2. Start Angular DevTools and select the **Components** tab.
3. Select the `app-product-list` component from the component tree:

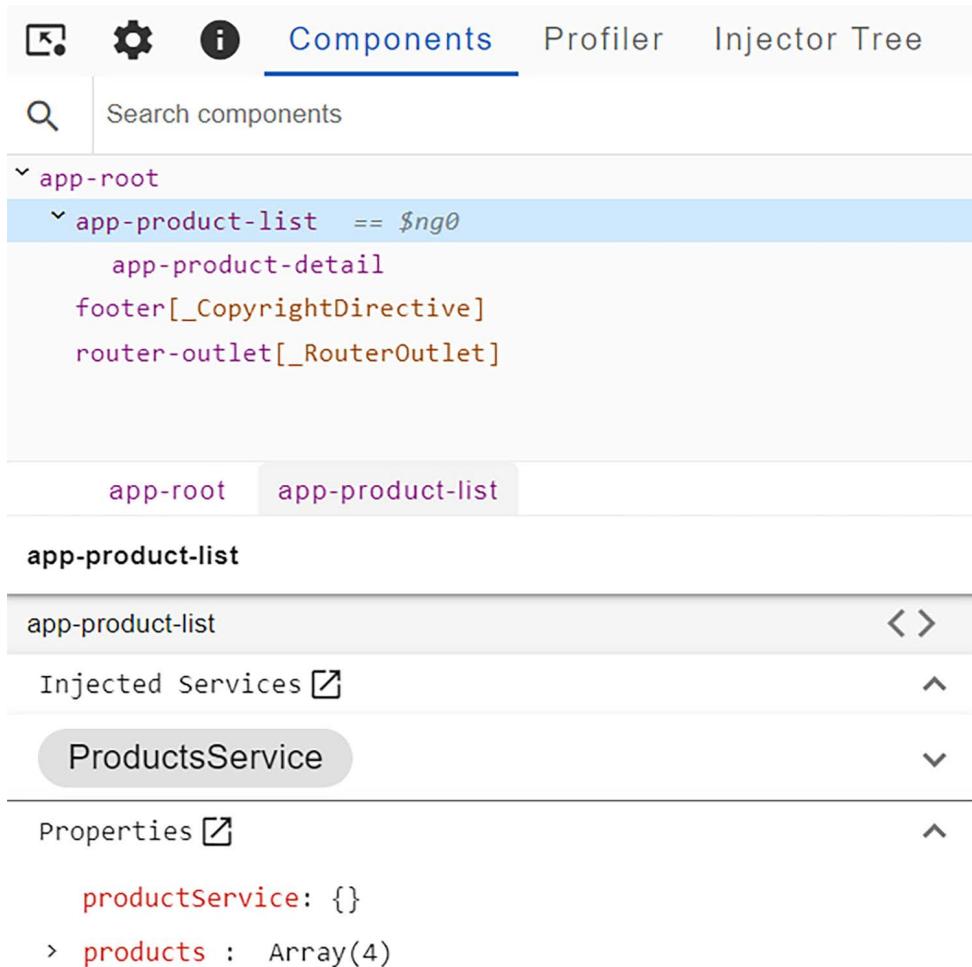


Figure 5.2: Components tab

In the preceding image, the **Injected Services** section contains the services injected in the component.

4. Click on the down arrow next to the **ProductsService** label, and you will see the following diagram:

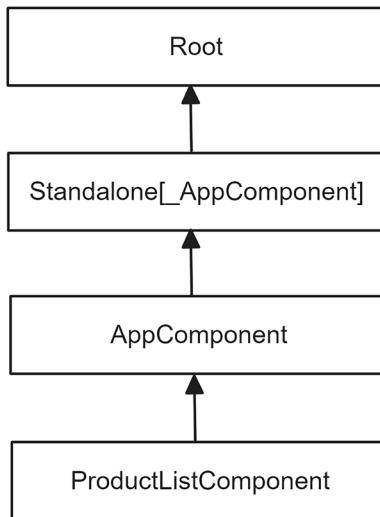


Figure 5.3: Product list injector hierarchy



The injector hierarchy diagram in Angular DevTools is in horizontal orientation. Here, we show it vertically for readability..

The preceding diagram depicts the injector hierarchy of the product list component. It contains two main injector hierarchy types common to an Angular application: **environment** and **element** injectors.

Environment injectors are configured using the `providedIn` property and the `providers` array in the application configuration object. In our case, we see the **Root** and **Standalone[_AppComponent]** injectors because the products service is provided from the root injector using the `providedIn` property.

Angular creates an element injector for each component which can be configured from the `providers` array of the `@Component` decorator, as we will see in the following section. In our case, we see the **AppComponent** and **ProductListComponent** injectors because these components are related directly to the product list.



You can select the **Injector Tree** tab of Angular DevTools for a more detailed analysis of the application injector hierarchy per type. You can also learn more about the different kinds of injectors at <https://angular.dev/guide/di/hierarchical-dependency-injection#types-of-injector-hierarchies>.

Components create injectors, so they are immediately available to their child components. We'll learn about this in detail in the following section.

Injecting services in the component tree

As we learned in the preceding section, Angular uses an element injector to provide services in components through the `providers` property of the `@Component` decorator. A service that registers with the element injector can serve two purposes:

- It can be shared with its child components
- It can create multiple copies of the service every time the component that provides the service is rendered

In the following sections, we'll learn how to apply each approach.

Sharing dependencies through components

A service provided through a component can be shared among the child components of the parent component, and it is immediately available for injection into their constructors. Child components reuse the same instance of the service as the parent component. Let's walk our way through an example to understand this better:

1. Create a new Angular component named `favorites`:

```
ng generate component favorites
```

2. Open the `favorites.component.ts` file and modify the import statements accordingly:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

3. Modify the `FavoritesComponent` class to use the `ProductsService` class and get the product list in a `products` component property:

```
export class FavoritesComponent implements OnInit {  
  products: Product[] = [];  
  
  constructor(private productService: ProductsService) {}  
  
  ngOnInit(): void {  
    this.products = this.productService.getProducts();  
  }  
}
```

4. Open the `favorites.component.html` file and replace its content with the following HTML code:

```
<ul class="pill-group">  
  @for (product of products | slice:1:3; track product.id) {  
    <li class="pill">  
      ★ {{product.title}}  
    </li>  
  }  
</ul>
```

In the preceding snippet, we iterate over the `products` array and use the `slice` pipe to display only two products.

5. Modify the `favorites.component.ts` file so that it imports the `CommonModule` class that is needed for the `slice` pipe:

```
import { CommonModule } from '@angular/common';  
import { Component, OnInit } from '@angular/core';  
import { Product } from '../product';  
import { ProductsService } from '../products.service';  
  
@Component({  
  selector: 'app-favorites',  
  imports: [CommonModule],  
  templateUrl: './favorites.component.html',
```

```
    styleUrl: './favorites.component.css'  
  })
```

6. Open the `favorites.component.css` file to add some CSS styles to our favorite products:

```
.pill-group {  
  display: flex;  
  flex-direction: column;  
  align-items: start;  
  flex-wrap: wrap;  
  gap: 1.25rem;  
}  
  
.pill {  
  display: flex;  
  align-items: center;  
  --pill-accent: var(--hot-red);  
  background: color-mix(in srgb, var(--hot-red) 5%, transparent);  
  color: var(--pill-accent);  
  padding-inline: 0.75rem;  
  padding-block: 0.375rem;  
  border-radius: 2.75rem;  
  border: 0;  
  transition: background 0.3s ease;  
  font-family: var(--inter-font);  
  font-size: 0.875rem;  
  font-style: normal;  
  font-weight: 500;  
  line-height: 1.4rem;  
  letter-spacing: -0.00875rem;  
  text-decoration: none;  
}
```

7. Open the `product-list.component.ts` file, import the `FavoritesComponent` class, and add the `ProductsService` class to the `providers` array of the `@Component` decorator:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
import { SortPipe } from '../sort.pipe';
import { ProductsService } from '../products.service';
import { FavoritesComponent } from '../favorites/favorites.
component';

@Component({
  selector: 'app-product-list',
  imports: [ProductDetailComponent, SortPipe, FavoritesComponent],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css'],
  providers: [ProductsService]
})
```

8. Open the `products.service.ts` file and remove the `providedIn` property from the `@Injectable` decorator since the element injector of the product list component will provide it.
9. Finally, open the `product-list.component.html` file and add the following HTML snippet to display the contents of the favorites component:

```
<h1>Favorites</h1>
<app-favorites></app-favorites>
```

When running the application using `ng serve`, you should see the following output:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Favorites

Microphone

Web camera

Copyright ©2024 All Rights Reserved

Figure 5.4: Product list with favorites

Let's explain what we did in the previous example in more detail. We injected `ProductsService` in `FavoritesComponent` but we did not provide it through its injector. So, how was the component aware of how to create an instance of the `ProductsService` class and use it? It wasn't. When we added the `favorites` component to the `ProductListComponent` template, we made it a direct child of this component, thus giving it access to all its provided services. In a nutshell, `FavoritesComponent` can use `ProductsService` out of the box because it is already provided through the element injector of its parent component, `ProductListComponent`.

So, even if `ProductsService` was initially registered with the environment root injector, we could also register it with the element injector of `ProductListComponent`. In the next section, we'll investigate how it is possible to achieve such behavior.

Root and component injectors

We have already learned that when we create an Angular service using the Angular CLI, the service is provided in the application's root injector by default. How does this differ when providing a service through the element injector of a component?

Services provided with the application root injector are available throughout the whole application. When a component wants to use such a service, it only needs to inject it, nothing more. Now, if the component provides the same service through its injector, it will get an instance of the service entirely different from the one from the root injector. This technique is called **service scope limiting** because we limit the scope of the service to a specific part of the component tree:

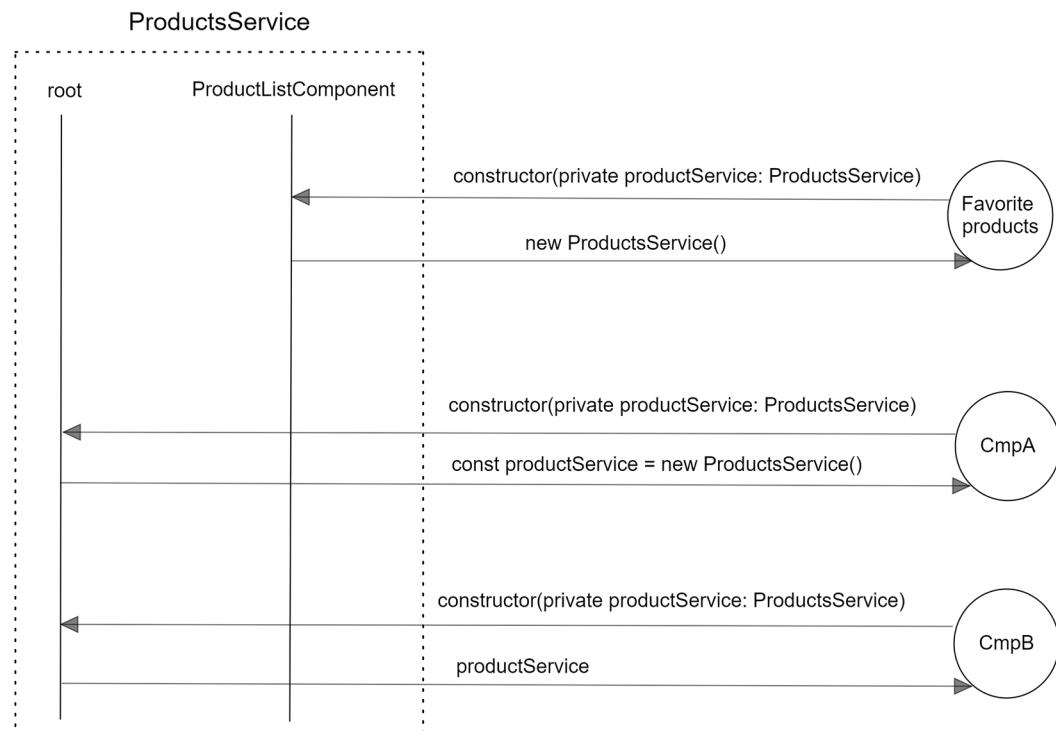


Figure 5.5: Service scope limiting

The previous diagram shows that `ProductsService` can be provided through two injectors: the application root injector and the element injector of the product list component. The `FavoritesComponent` class injects `ProductsService` to use it. As we have already seen, `FavoritesComponent` is a child component of `ProductListComponent`.

According to the injector hierarchy, it will first ask the injector of its parent component, `ProductListComponent`, about providing the service. The `ProductListComponent` class indeed provides `ProductsService`, so it creates a new instance of the service and returns it to `FavoritesComponent`.

Now, consider that another component in our application, `CmpA`, wants to use `ProductsService`. Since it is not a child component of `ProductListComponent` and does not contain any parent component that provides the required service, it will finally reach the application root injector. The root injector that provides `ProductsService` checks if it has already created an instance for that service. If not, it creates a new one, called `productService`, and returns it to `CmpA`. It also keeps `productService` in the local pool of services for later use.

Suppose another component, `CmpB`, wants to use `ProductsService` and asks the application root injector. The root injector knows it has already created the `productService` instance when `CmpA` requested it and returns it immediately to the `CmpB` component.

Sandboxing components with multiple instances

When we provide a service through the element injector and inject it into the component's constructor, a new instance is created every time the component is rendered on the page. It can come in handy in cases such as when we want to have a local cache service for each component. We will explore this scenario by transforming our Angular application so that the product list displays a quick view of each product using an Angular service:

1. Run the following command to create a new Angular component for the product view:

```
ng generate component product-view
```

2. Open the `product-view.component.ts` file and declare an input property named `id` so we can pass a unique identifier of the product we want to display:

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

@Component({
  selector: 'app-product-view',
  imports: [],
})
```

```

        templateUrl: './product-view.component.html',
        styleUrls: ['./product-view.component.css']
    })
export class ProductViewComponent {
    id = input<number>();
}

```

- Run the following Angular CLI command inside the `product-view` folder to create an Angular service that will be dedicated to the product view component:

```
ng generate service product-view
```

- Open the `product-view.service.ts` file and remove the `providedIn` property from the `@Injectable` decorator because we will provide it later in the product view component.
- Inject `ProductsService` into the constructor of the `ProductViewService` class:

```

import { Injectable } from '@angular/core';
import { ProductsService } from '../products.service';

@Injectable()
export class ProductViewService {

    constructor(private productService: ProductsService) { }
}

```

The preceding technique is called **service-in-a-service** because we inject one Angular service into another.

- Create a method named `getProduct` that takes an `id` property as a parameter. The method will call the `getProducts` method of the `ProductsService` class and search through the product list based on the `id`. If it finds the product, it will keep it in a local variable named `product`:

```

import { Injectable } from '@angular/core';
import { ProductsService } from '../products.service';
import { Product } from '../product';

@Injectable()
export class ProductViewService {
    private product: Product | undefined;

```

```
constructor(private productService: ProductsService) { }

getProduct(id: number): Product | undefined {
  const products = this.productService.getProducts();
  if (!this.product) {
    this.product = products.find(product => product.id === id)
  }
  return this.product;
}
}
```

We have already created the essential Angular artifacts for working with the product view component. All we need to do now is connect them and wire them up to the product list:

1. Inject `ProductViewService` in the constructor of the `ProductViewComponent` and implement the `ngOnInit` method:

```
import { Component, input, OnInit } from '@angular/core';
import { ProductViewService } from './product-view.service';

@Component({
  selector: 'app-product-view',
  imports: [],
  templateUrl: './product-view.component.html',
  styleUrls: ['./product-view.component.css'],
  providers: [ProductViewService]
})
export class ProductViewComponent implements OnInit {
  id = input<number>();

  constructor(private productViewService: ProductViewService) {}

  ngOnInit(): void {
  }
}
```

2. Create a component property to keep the product that we will fetch from the ProductViewService class:

```

import { Component, input, OnInit } from '@angular/core';
import { ProductViewService } from './product-view.service';
import { Product } from '../product';

@Component({
  selector: 'app-product-view',
  imports: [],
  templateUrl: './product-view.component.html',
  styleUrls: ['./product-view.component.css'],
  providers: [ProductViewService]
})
export class ProductViewComponent implements OnInit {
  id = input<number>();
  product: Product | undefined;

  constructor(private productViewService: ProductViewService) {}

  ngOnInit(): void {
  }
}

```

3. Modify the ngOnInit method so that it calls the getProduct method of the ProductViewService class as follows:

```

ngOnInit(): void {
  this.product = this.productViewService.getProduct(this.id()!);
}

```

In the preceding snippet, we pass the id component property to the getProduct method as a parameter and assign the returned value to the product property.

4. Open the product-view.component.html file and replace its content with the following HTML template:

```

@switch (product?.title) {
  @case ('Keyboard') { 🖱️ }

```

```
@case ('Microphone') { 🎤 }
@default { 🎵 }
}
{{product?.title}}
```

5. Open the `product-list.component.ts` file and import the `ProductViewComponent` class:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
import { SortPipe } from '../sort.pipe';
import { ProductsService } from '../products.service';
import { ProductViewComponent } from '../product-view/product-view.
component';

@Component({
  selector: 'app-product-list',
  imports: [ProductDetailComponent, SortPipe, ProductViewComponent],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

6. Finally, open the `product-list.component.html` file and modify the `@for` block to use the product view component:

```
<ul class="pill-group">
  @for (product of products | sort; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      <app-product-view [id]="product.id"></app-product-view>
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

If we run our application with the `ng serve` command, we will see that the product list is still displayed correctly.

Each rendered product view component creates a dedicated sandboxed `ProductViewService` instance for its purpose. Any other component cannot share the instance or be changed except by the component that provides it.

Try to provide `ProductViewService` in `ProductListComponent` instead of `ProductViewComponent`; you will see that only one product is rendered multiple times:

Products (4)

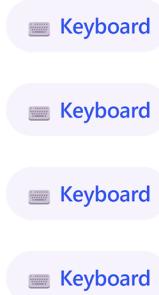


Figure 5.6: Product list

In this case, only one service instance is shared among the child components. Why is that? Recall the business logic of the `getProduct` method from the `ProductViewService` class:

```
getProduct(id: number): Product | undefined {
  const products = this.productService.getProducts();
  if (!this.product) {
    this.product = products.find(product => product.id === id)
  }
  return this.product;
}
```

In the preceding method, the `product` property is set initially when we provide the service inside `ProductListComponent`. Since we have only one instance of the service, the value of the property will remain the same while we render the product view component multiple times.

We've learned how dependencies are injected into the component hierarchy and how provider lookup is performed by bubbling the request upward in the component tree. However, what if we want to constrain such injection or lookup actions? We'll see how to do so in the next section.

Restricting provider lookup

We can only constrain dependency lookup to the next upper level. To do so, we need to apply the `@Host` decorator to those dependency parameters whose provider lookup we want to restrict:

```
import { CommonModule } from '@angular/common';
import { Component, Host, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';

@Component({
  selector: 'app-favorites',
  imports: [CommonModule],
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css'
})
export class FavoritesComponent implements OnInit {
  products: Product[] = [];

  constructor(@Host() private productService: ProductsService) {}

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }
}
```

In the preceding example, the element injector of `FavoritesComponent` will look for the `ProductsService` class in its providers. If it does not provide the service, it will not bubble up the injector hierarchy; instead, it will stop and throw an exception in the console window of the browser:

Error: NG0201: No provider for _ProductsService found in NodeInjector.

We can configure the injector so that it does not throw an error if we decorate the service with the `@Optional` decorator:

```
import { CommonModule } from '@angular/common';
import { Component, Host, OnInit, Optional } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

```
@Component({
  selector: 'app-favorites',
  imports: [CommonModule],
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css']
})
export class FavoritesComponent implements OnInit {
  products: Product[] = [];

  constructor(@Optional() @Host() private productService: ProductsService) {}

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }
}
```

However, using the `@Optional` decorator does not solve the actual problem. The preceding snippet will still throw an error, different than the previous one, because we still use the `@Host` decorator that limits searching the `ProductsService` class in the injector hierarchy. We need to refactor the `ngOnInit` lifecycle hook event so that it takes care of not finding the service instance.

The `@Host` and `@Optional` decorators define the level at which the injector searches for dependencies. There are two other decorators, called `@Self` and `@SkipSelf`. When using the `@Self` decorator, the injector looks for dependencies in the injector of the current component. On the contrary, the `@SkipSelf` decorator instructs the injector to skip the local injector and search further up in the injector hierarchy.



The `@Host` and `@Self` decorators work similarly. For more information about when to use each, have a look at <https://angular.dev/guide/di/hierarchical-dependency-injection#self> and <https://angular.dev/guide/di/hierarchical-dependency-injection#host>.

So far, we have learned how the Angular DI framework uses classes as dependency tokens to work out the type required and return it from any providers available in the injector hierarchy. However, there are cases where we might need to override the instance of a class or provide types that are not actual classes, such as primitive types.

Overriding providers in the injector hierarchy

We already learned how to use the `providers` array of the `@Component` decorator in the *Sharing dependencies through components* section:

```
  providers: [ProductsService]
```

The preceding syntax is called **class provider** syntax and is shorthand for the **provide object literal** syntax shown below:

```
  providers: [
    { provide: ProductsService, useClass: ProductsService }
  ]
```

The preceding syntax uses an object with the following properties:

- `provide`: This is the token used to configure the injector. It is the actual class that consumers of the dependency inject into their constructors.
- `useClass`: This is the actual implementation the injector will provide to the consumers. The property name will differ according to the implementation type provided. The type can be a class, a value, or a factory function. In this case, we use `useClass` because we are providing a class.

Let's look at some examples to get an overview of how to use the `provide` object literal syntax.

Overriding service implementation

We have already learned that a component could share its dependencies with its child components. Consider the `FavoritesComponent`, where we used the `slice` pipe to display a list of favorite products in its template. What if it needs to get data through a trimmed version of `ProductsService` and not directly from the service instance of `ProductListComponent`? We could create a new service extending the `ProductsService` class and filtering out data using the native `Array.slice` method. Let's create the new service and learn how to use it:

1. Run the following command to generate the service:

```
ng generate service favorites
```

2. Open the `favorites.service.ts` file and add the following import statements:

```
import { Product } from './product';
import { ProductsService } from './products.service';
```

3. Use the `extends` keyword in the class definition to indicate that `ProductsService` is the base class of `FavoritesService`:

```
export class FavoritesService extends ProductsService {  
  
    constructor() { }  
}
```

4. Modify the constructor to call the `super` method and execute any business logic inside the base class constructor:

```
constructor() {  
    super();  
}
```

5. Create the following service method that uses the `slice` method to return only the first two products from the list:

```
override getProducts(): Product[] {  
    return super.getProducts().slice(1, 3);  
}
```

The preceding method is marked with the `override` keyword to indicate that the implementation of the method replaces the corresponding method of the base class.

6. Open the `favorites.component.ts` file and add the following `import` statement:

```
import { FavoritesService } from './favorites.service';
```

7. Add the `FavoritesService` class in the `providers` array of the `@Component` decorator as follows:

```
@Component({  
    selector: 'app-favorites',  
    imports: [],  
    templateUrl: './favorites.component.html',  
    styleUrls: ['./favorites.component.css'],  
    providers: [  
        { provide: ProductsService, useClass: FavoritesService }  
    ]  
})
```

In the preceding snippet, we removed `CommonModule` from the `imports` array because we no longer need the `slice` pipe.

- Finally, open the `favorites.component.html` file and remove the `slice` pipe from the `@for` block.

If we run the application using the `ng serve` command, we will see that the **Favorites** section is still displayed correctly:

Favorites

★ Microphone

★ Web camera

Figure 5.7: Favorite products list



The preceding output assumes that you have already imported and added the `favorites` component in the `product-list` component.

The `useClass` property essentially overwrote the initial implementation of the `ProductsService` class for the `favorites` component. Alternatively, we can go the extra mile and use a function to return a specific object instance that we need, as we will learn in the following section.

Providing services conditionally

In the example in the previous section, we used the `useClass` syntax to replace the implementation of the injected `ProductsService` class. Alternatively, we could create a **factory function** that decides whether it will return an instance of the `FavoritesService` or `ProductsService` class according to a condition. The function would reside in a simple TypeScript file named `favorites.ts`:

```
import { FavoritesService } from './favorites.service';
import { ProductsService } from './products.service';

export function favoritesFactory(isFavorite: boolean) {
  return () => {
    if (isFavorite) {
      return new FavoritesService();
    }
    return new ProductsService();
  }
}
```

```
        return new FavoritesService();
    }
    return new ProductsService();
}
}
```

We could then modify the providers array in the `favorites.component.ts` file as follows:

```
import { CommonModule } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
import { favoritesFactory } from '../favorites';

@Component({
  selector: 'app-favorites',
  imports: [CommonModule],
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css'],
  providers: [
    { provide: ProductsService, useFactory: favoritesFactory(true) }
  ]
})
)
```

It is worth noting that if one of the services also injected other dependencies, the previous syntax would not suffice. For example, if the `FavoritesService` class was dependent on the `ProductViewService` class, we would add it to the `deps` property of the `provide` object literal syntax:

```
providers: [
  {
    provide: ProductsService,
    useFactory: favoritesFactory(true),
    deps: [ProductViewService]
  }
]
```

We could then use it in the factory function of the `favorites.ts` file as follows:

```
export function favoritesFactory(isFavorite: boolean) {
  return (productViewService: ProductViewService) => {
    if (isFavorite) {
      return new FavoritesService();
    }
    return new ProductsService();
  };
}
```

We have already learned how to provide an alternate class implementation for an Angular service. What if the dependency we want to provide is not a class but a string or an object? We can use the `useValue` syntax to accomplish this task.

Transforming objects in Angular services

It is common to keep application settings in a constant object in real-world applications. How could we use the `useValue` syntax to provide these settings in our components? We will learn more by creating settings for our application, such as the version number and the title:

1. Create an `app.settings.ts` file in the `src\app` folder of the Angular CLI workspace and add the following contents:

```
export interface AppSettings {
  title: string;
  version: string;
}

export const appSettings: AppSettings = {
  title: 'My e-shop',
  version: '1.0'
};
```

You may think we could provide these settings as `{ provide: AppSettings, useValue: appSettings }`, but this would throw an error because `AppSettings` is an interface, not a class. Interfaces are syntactic sugar in TypeScript that are thrown away during compilation. Instead, we should provide an `InjectionToken` object.

2. Add the following statement to import the `InjectionToken` class from the `@angular/core` npm package:

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

3. Declare the following constant variable that uses the `InjectionToken` type:

```
export const APP_SETTINGS = new InjectionToken<AppSettings>('app.settings');
```

4. Open the `app.component.ts` file and modify the `import` statements as follows:

```
import { Component, inject } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';
import { CopyrightDirective } from './copyright.directive';
import { APP_SETTINGS, appSettings } from './app.settings';
```

5. Add the application settings token in the `providers` array of the `@Component` decorator:

```
@Component({
  selector: 'app-root',
  imports: [
    RouterOutlet,
    ProductListComponent,
    CopyrightDirective
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    { provide: APP_SETTINGS, useValue: appSettings }
  ]
})
```



The `useValue` syntax is particularly useful when testing Angular applications. We will use it extensively when we learn about unit testing in *Chapter 13, Unit Testing Angular Applications*.

6. Add the following property in the AppComponent class:

```
settings = inject(APP_SETTINGS);
```

7. Open the app.component.html file and modify the <footer> tag to include the application version:

```
<footer appCopyright> - v{{ settings.version }}</footer>
```

8. Run the application using the ng serve command and observe the footer in the application output:

Copyright ©2024 All Rights Reserved - v1.0

Note that although the AppSettings interface did not play a significant role in the injection process, we need it to provide typing on the configuration object.

Angular DI is a powerful and robust mechanism that allows us to manage the dependencies of our applications efficiently. The Angular team has put much effort into making it simple to use and removed the burden from the developer's side. As we have seen, the combinations are plentiful, and how we will use them depends on the use case.

Summary

The Angular DI implementation is the backbone of the Angular framework. Angular components delegate complex tasks to Angular services, based on the Angular DI.

In this chapter, we learned what Angular DI is and how to leverage it by creating Angular services. We explored different ways of injecting Angular services into components. We saw how to share services between components, isolate services in components, and define dependency access through the component tree.

Finally, we investigated how to override Angular services by replacing the service implementation or transforming existing objects into services.

In the next chapter, we will learn what reactive programming is and how we can use observables in the context of Angular applications.

6

Reactive Patterns in Angular

Handling asynchronous information is a common task in our everyday lives as developers. **Reactive programming** is a paradigm that helps us consume, digest, and transform asynchronous information using data streams. **RxJS** is a JavaScript library that provides methods to manipulate data streams using **observables**.

Angular provides an unparalleled toolset to help us when working with asynchronous data. Observable streams are at the forefront of this toolset, giving developers a rich set of capabilities when creating Angular applications. The core of the Angular framework is lightly dependent on RxJS. Other Angular packages, such as the router and the HTTP client, are more tightly coupled with observables. However, at the time of writing, the Angular team is currently investigating making the preceding packages less dependent on observables.

In this chapter, we will learn about the following concepts:

- Strategies for handling asynchronous information
- Reactive programming in Angular
- The RxJS library
- Subscribing to observables
- Unsubscribing from observables

Technical requirements

The chapter contains various code samples to walk you through observables and RxJS. You can find the related source code in the ch06 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Strategies for handling asynchronous information

We manage data asynchronously in different forms, such as consuming data from a backend API, a typical operation in our daily development workflow, or reading contents from the local file system. We always consume data over HTTP, such as when authenticating users by sending credentials to an authentication service. We also use HTTP when fetching the latest posts in our favorite social network application.

Modern mobile devices have introduced a unique way of consuming remote services. They defer requests and response consumption until mobile connectivity is available. Responsivity and availability have become a big deal.

Although internet connections are high-speed nowadays, response time is always involved when serving such information. Thus, as we will see in this section, we put in place mechanisms to handle the state of our applications transparently for the end user.

Shifting from callback hell to promises

Sometimes we might need to build functionalities in our application that change its state asynchronously once time has elapsed. In these cases we must introduce code patterns, such as the **callback pattern**, to handle this deferred change in the application state.

In a callback, the function that triggers asynchronous action accepts another function as a parameter. The function is executed when the asynchronous operation has been completed.

You will need the source code of the Angular application we created in *Chapter 5, Managing Complex Tasks with Services*, to follow along with the rest of the chapter. After you get the code, we suggest you take the following actions for simplicity:



- Remove the favorites folder
- Remove the favorites.service.ts and its unit test file
- Remove the favorite.ts file
- Remove the numeric.directive.ts file and its unit test file
- Remove the product-view folder

Let's see how to use a callback through an example:

1. Open the app.component.html file and add a <header> HTML element to display the title component property using interpolation:

```
<header>{{ title }}</header>
<main class="main">
  <div class="content">
    <app-product-list></app-product-list>
  </div>
</main>
<footer appCopyright> - v{{ settings.version }}</footer>
<router-outlet />
```

2. Open the `app.component.ts` file and create the following property:

```
private setTitle = () => {
  this.title = this.settings.title;
}
```

The `setTitle` property is used to change the `title` component property based on the `title` property from application settings. It returns an arrow function because we will use it as a callback to another method.

3. Next, create a `changeTitle` method that calls another method, named, by convention, `callback`, after two seconds:

```
private changeTitle(callback: Function) {
  setTimeout(() => {
    callback();
  }, 2000);
}
```

4. Add a `constructor` to call the `changeTitle` method, passing the `setTitle` property as a parameter:

```
constructor() {
  this.changeTitle(this.setTitle);
}
```

In the preceding snippet, we use the `setTitle` property without parentheses because we pass function signatures and not actual function calls when we use callbacks.

If we run the Angular application using the `ng serve` command, we see that the `title` property changes after two seconds. The problem with the pattern we just described is that the code can become confusing and cumbersome as we introduce more nested callbacks.

Consider the following scenario where we need to drill down into a folder hierarchy to access photos on a device:

```
getRootFolder(folder => {
  getAssetsFolder(folder, assets => {
    getPhotos(assets, photos => {});
  });
});
```

We depend on the previous asynchronous call and the data it brings back before we can do the next call. We must execute a method inside a callback that executes another method with a callback. The code quickly looks complex and difficult to read, leading to a situation known as **callback hell**.

We can avoid callback hell using **promises**. Promises introduce a new way of envisioning asynchronous data management by conforming to a neater and more solid interface. Different asynchronous operations can be chained at the same level and even be split and returned from other functions.

To better understand how promises work, let's refactor our previous callback example:

1. Create a new method in the `AppComponent` class named `onComplete` that returns a `Promise` object. A promise can either be **resolved** or **rejected**. The `resolve` parameter indicates that the promise was completed successfully and optionally returns a result:

```
private onComplete() {
  return new Promise<void>(resolve => {
  });
}
```

2. Introduce a timeout of two seconds in the promise so that it resolves after this time has elapsed:

```
private onComplete() {
  return new Promise<void>(resolve => {
    setTimeout(() => {
      resolve();
    }, 2000);
  });
}
```

3. Now, replace the `changeTitle` call in the constructor with the promise-based method. To execute a method that returns a promise, we invoke the method and chain it with the `then` method:

```
constructor() {  
    this.onComplete().then(this.setTitle);  
}
```

We should not notice any significant difference if we rerun the Angular application. The real value of promises lies in the simplicity and readability afforded to our code. We could now refactor the previous folder hierarchy example accordingly:

```
getRootFolder()  
    .then(getAssetsFolder)  
    .then(getPhotos);
```

The chaining of the `then` method in the preceding code shows how we can line up one asynchronous call after another. Each previous asynchronous call passes its result in the upcoming asynchronous method.

Promises are compelling, but sometimes we might need to produce a response output that follows a more complex digest process or even cancel the whole process. We cannot accomplish such behavior with promises because they are triggered as soon as they are instantiated. In other words, promises are not lazy. On the other hand, the possibility of tearing down an asynchronous operation after it has been fired but not completed yet can become quite handy in specific scenarios. Promises allow us to resolve or reject an asynchronous operation, but sometimes we might want to abort everything before getting to that point.

On top of that, promises behave as one-time operations. Once they are resolved, we cannot expect to receive any further information or state change notifications unless we run everything from scratch. To summarize the limitations of promises:

- They cannot be canceled
- They are immediately executed
- They are one-time operations; there is no easy way to retry them
- They respond with only one value

Let's illustrate some of the limitations with an example:

1. Replace `setTimeout` with `setInterval` in the `onComplete` method:

```
private onComplete() {
  return new Promise<void>(resolve => {
    setInterval(() => {
      resolve();
    }, 2000);
  });
}
```

The promise will now resolve repeatedly every two seconds.

2. Modify the `setTitle` property to append the current `timestamp` in the `title` property of the component:

```
private setTitle = () => {
  const timestamp = new Date();
  this.title = `${this.settings.title} ${timestamp}`;
}
```

3. Run the Angular application and you will notice that the timestamp is set only once after two seconds and never changes again. The promise resolves itself, and the entire asynchronous event terminates at that very moment.

We may need a more proactive implementation of asynchronous data handling to fix the preceding behavior, which is where observables come into the picture.

Observables in a nutshell

An observable is an object that maintains a list of dependents, called **observers**, and informs them about state and data changes by emitting events asynchronously. To do so, the observable implements all the necessary machinery to produce and emit such events. It can be triggered and canceled at any time, regardless of whether it has emitted the expected data already.

Observers must subscribe to an observable to be notified and react to reflect the state change. This pattern, known as the **observer pattern**, allows concurrent operations and more advanced logic. These observers, also known as **subscribers**, keep listening to whatever happens in the observable until it is destroyed. We can see all this with more transparency in an actual example:

1. Import the Observable artifact from the rxjs npm package:

```
import { Observable } from 'rxjs';
```

2. Create a component property named title\$ that creates an Observable object. The constructor of an observable accepts an observer object as a parameter. The observer is an arrow function that contains the business logic that will be executed when someone uses the observable. Call the next method of the observer every two seconds to indicate a data or application state change:

```
title$ = new Observable(observer => {
  setInterval(() => {
    observer.next();
  }, 2000);
});
```



When we define an observable variable, we tend to append the \$ sign to the variable name. It is a convention that we follow to identify observables in our code efficiently and quickly.

3. Modify the constructor component to use the newly created title\$ property:

```
constructor() {
  this.title$.subscribe(this.setTitle);
}
```

We use the subscribe method to register to the title\$ observable and get notified of any changes. If we do not call this method, the setTitle method will never execute.



An observable will not do anything unless a subscriber subscribes to it.

If you run the application, you will notice that the timestamp changes every two seconds. Congratulations! You have entered the world of observables and reactive programming!

Observables return a stream of events, and our subscribers receive prompt notifications of those events so that they can act accordingly. They do not perform an asynchronous operation and terminate (although we can configure them to do so) but start a stream of ongoing events to which we can subscribe.

That's not all, however. This stream can combine many operations before hitting observers subscribed to it. Just as we can manipulate arrays with methods such as `map` or `filter` to transform them, we can do the same with the stream of events emitted by observables. It is a pattern known as reactive programming, and Angular makes the most of this paradigm to handle asynchronous information.

Reactive programming in Angular

The observer pattern stands at the core of reactive programming. The most basic implementation of a reactive script encompasses several concepts that we need to become familiar with:

- An observable
- An observer
- A timeline
- A stream of events
- A set of composable operators

It may sound daunting, but it isn't. The big challenge here is to change our mindset and learn how to think reactively, which is the primary goal of this section.



Reactive programming entails applying asynchronous subscriptions and transformations to observable streams of events.

Let's explain through a more descriptive example. Think about an interaction device such as a keyboard. It has keys that the user presses. Each one of those keystrokes triggers a specific keyboard event, such as `keyUp`. The `keyUp` event features a wide range of metadata, including—but not limited to—the numeric code of the specific key the user pressed at a given moment. As the user continues hitting keys, more `keyUp` events are triggered and piped through an imaginary timeline. The timeline is a continuous stream of data where the `keyUp` event can happen at any time; after all, the user decides when to press those keys.

Recall the example with observables from the previous section. That code could notify an observer that every two seconds, another value was emitted. We know how often a timer interval is triggered. In the case of `keyUp` events, we don't know because they are not under our control. Let's try to explain it further by implementing a key logger in our application:

1. Create a new Angular component named key-logger:

```
ng generate component key-logger
```

2. Open the key-logger.component.html file and replace its content with the following HTML template:

```
<input type="text" #keyContainer />  
You pressed: {{keys}}
```

In the preceding template, we added an `<input>` HTML element and attached the `keyContainer` template reference variable.



A template reference variable can be added to any HTML element, not just components.

We also display a `keys` property representing all the keyboard keys the user has pressed.

3. Open the key-logger.component.ts file and import the `OnInit`, `viewChild`, and `ElementRef` artifacts from the `@angular/core` npm package:

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

4. Create the following properties in the `KeyLoggerComponent` class:

```
input = viewChild<ElementRef>('keyContainer');  
keys = '';
```

The `input` property is used to query the `<input>` HTML element using the `keyContainer` template reference variable.

5. Add the following `import` statement to import the `fromEvent` artifact from the `rxjs` npm package:

```
import { fromEvent } from 'rxjs';
```

The RxJS library has various helpful artifacts, called **operators**, that we can use with observables. The `fromEvent` operator creates an observable from the DOM event of a native HTML element.

6. Implement the `ngOnInit` method from the `OnInit` interface to listen for keyup events in the `<input>` element and save pressed keys in the `keys` property:

```
export class KeyLoggerComponent implements OnInit {
  input = viewChild<ElementRef>('keyContainer');
  keys = '';

  ngOnInit(): void {
    const logger$ = fromEvent<KeyboardEvent>(this.input()!.nativeElement, 'keyup');
    logger$.subscribe(evt => this.keys += evt.key);
  }
}
```

Notice that we get access to the native HTML input element through the `nativeElement` property of the template reference variable. The result of querying using the `viewChild` function is an `ElementRef` object, which is a wrapper over the actual HTML element.

7. Open the `app.component.ts` file and import the `KeyLoggerComponent` class:

```
import { Component, inject } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';
import { CopyrightDirective } from './copyright.directive';
import { APP_SETTINGS, appSettings } from './app.settings';
import { Observable } from 'rxjs';
import { KeyLoggerComponent } from './key-logger/key-logger.component';

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

```
    providers: [
      { provide: APP_SETTINGS, useValue: appSettings }
    ]
})
```

8. Open the `app.component.html` file and add the `<app-key-logger>` selector in the template:

```
<header>{{ title }}</header>
<main class="main">
  <div class="content">
    <app-product-list></app-product-list>
  </div>
</main>
<footer appCopyright> - v{{ settings.version }}</footer>
<router-outlet />
<app-key-logger></app-key-logger>
```

Run the application using the `ng serve` command and start pressing keys to verify the usage of the key logger that we have just created:

angular

You pressed: angular

Figure 6.1: Key logger output

An essential aspect of observables is using operators and chaining observables together, enabling **rich composition**. Observable operators look like array methods when we want to use them. For example, a `map` operator for observables is used similarly to the `map` method of an array. In the following section, we will learn about the RxJS library, which provides these operators, and learn about some of them through examples.

The RxJS library

As mentioned previously, Angular comes with a peer dependency on RxJS, the JavaScript flavor of the **ReactiveX** library, which allows us to create observables out of a large variety of scenarios, including the following:

- Interaction events
- Promises
- Callback functions
- Events

Reactive programming does not aim to replace asynchronous patterns like promises or callbacks. All the way around, it can leverage them as well to create observable sequences.

RxJS has built-in support for various composable operators to transform, filter, and combine the resulting event streams. Its API provides convenient methods for observers to subscribe to these streams so that our components can respond accordingly to state changes or input interaction. Let's see some of these operators in action in the following subsections.

Creating observables

We have already learned how to create an observable from a DOM event using the `fromEvent` operator. Two other popular operators concerned with observable creation are the `of` and `from` operators.

The `of` operator is used to create an observable from values such as numbers:

```
const values = of(1, 2, 3);
values.subscribe(value => console.log(value));
```

The previous snippet prints the numbers **1**, **2**, and **3** in the browser console window *in order*.

The `from` operator is used to convert an array to an observable:

```
const values = from([1, 2, 3]);
values.subscribe(value => console.log(value));
```

The `from` operator is also very useful when converting promises or callbacks to observables. We could wrap the `onComplete` method in the `constructor` of the `AppComponent` class as follows:

```
constructor() {
  const complete$ = from(this.onComplete());
  complete$.subscribe(this.setTitle);
}
```



The `from` operator is an excellent way to migrate to observables if you use promises in an existing application!

Besides creating observables, the RxJS library also contains a couple of handy operators to manipulate and transform data emitted from observables.

Transforming observables

We have already learned how to create a numeric-only directive in *Chapter 4, Enriching Applications Using Pipes and Directives*. We will now use RxJS operators to accomplish the same thing in our key logger component:

1. Open the `key-logger.component.ts` file and import the `tap` operator from the `rxjs` npm package:

```
import { fromEvent, tap } from 'rxjs';
```

2. Refactor the `ngOnInit` method as follows:

```
ngOnInit(): void {
  const logger$ = fromEvent<KeyboardEvent>(this.input()!.nativeElement, 'keyup');
  logger$.pipe(
    tap(evt => this.keys += evt.key)
  ).subscribe();
}
```

The pipe operator links and combines multiple operators separated by commas. We can think of it as a recipe that defines the operators that should be applied to an observable. One of them is the `tap` operator, which is used when we want to do something with the data emitted without modifying it.

3. We want to exclude non-numeric values that the `logger$` observable emits. We already get the actual key pressed from the `evt` property, but it returns alphanumeric values. It would not be efficient to list all non-numeric values and exclude them manually. Instead, we will use the `map` operator to get the actual Unicode value of the key. It behaves similarly to the `map` method of an array as it returns an observable with a modified version of the initial data. Import the `map` operator from the `rxjs` npm package:

```
import { fromEvent, tap, map } from 'rxjs';
```

4. Add the following snippet above the `tap` operator in the `ngOnInit` method:

```
map(evt => evt.key.charCodeAt(0))
```

5. We can now add the `filter` operator, which operates similarly to the `filter` method of an array for excluding non-numeric values. Import the `filter` operator from the `rxjs` npm package:

```
import { fromEvent, tap, map, filter } from 'rxjs';
```

- Add the following snippet after the `map` operator in the `ngOnInit` method:

```
filter(code => (code > 31 && (code < 48 || code > 57)) === false)
```

- The observable currently emits Unicode character codes. We must convert them back to keyboard characters to display them on the HTML template. Refactor the `tap` operator to accommodate this change:

```
tap(digit => this.keys += String.fromCharCode(digit))
```

As a final touch, we will add an input binding in the component to toggle the numeric-only feature on and off conditionally:

- Add the `input` function in the `import` statement of the `@angular/core` npm package:

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

- Add a `numeric` input property in the `KeyLoggerComponent` class:

```
numeric = input(false);
```

- Refactor the `filter` operator in the `ngOnInit` method so that it takes into account the `numeric` property:

```
filter(code => {
  if (this.numeric()) {
    return (code > 31 && (code < 48 || code > 57)) === false;
  }
  return true;
})
```

The `logger$` observable will filter non-numeric values only if the `numeric` input property is `true`.

- The `ngOnInit` method should finally look like the following:

```
ngOnInit(): void {
  const logger$ = fromEvent<KeyboardEvent>(this.input()!.nativeElement, 'keyup');
  logger$.pipe(
    map(evt => evt.key.charCodeAt(0)),
    filter(code => {
      if (this.numeric()) {
```

```
        return (code > 31 && (code < 48 || code > 57)) === false;
    }
    return true;
}),
tap(digit => this.keys += String.fromCharCode(digit))
.subscribe();
}
```

5. Open the `app.component.html` file and add a binding to the `numeric` property in the `<app-key-logger>` selector:

```
<app-key-logger [numeric]="true"></app-key-logger>
```

6. Run the application using the `ng serve` command and enter `Angular 19` inside the input box:



```
Angular 19 You pressed: 19
```

Figure 6.2: Numeric key logger

We have seen RxJS operators manipulating observables that return primitive data types such as numbers, strings, and arrays. In the following section, we will learn how to use observables in our e-shop application.

Subscribing to observables

We have already learned that an observer needs to subscribe to an observable to get emitted data. The observer in our case will be the product list component and the observable will reside inside the `products.service.ts` file. Thus, we first need to convert the `ProductsService` class to use observables instead of plain arrays so that components can subscribe to get data:

1. Open the `products.service.ts` file and add the following import statement:

```
import { Observable, of } from 'rxjs';
```

2. Extract the product data used in the `getProducts` method into a separate service property to enhance code readability:

```
private products: Product[] = [
{
    id: 1,
    title: 'Keyboard',
    price: 100,
```

```
        categories: {
          1: 'Computing',
          2: 'Peripherals'
        }
      },
      {
        id: 2,
        title: 'Microphone',
        price: 35,
        categories: { 3: 'Multimedia' }
      },
      {
        id: 3,
        title: 'Web camera',
        price: 79,
        categories: {
          1: 'Computing',
          3: 'Multimedia'
        }
      },
      {
        id: 4,
        title: 'Tablet',
        price: 500,
        categories: { 4: 'Entertainment' }
      }
    ];
  
```

3. Modify the `getProducts` method so that it returns the `products` property as an observable:

```
getProducts(): Observable<Product[]> {
  return of(this.products);
}
```

In the preceding snippet, we use the `of` operator to create a new observable from the `products` array.

The `ProductsService` class now emits product data using observables. We must modify the component to subscribe and get this data:

1. Open the `product-list.component.ts` file and create a `getProducts` method in the `ProductListComponent` class:

```
private getProducts() {  
    this.productService.getProducts().subscribe(products => {  
        this.products = products;  
    });  
}
```

In the preceding method, we subscribe to the `getProducts` method of the `ProductsService` class because it returns an observable instead of a plain array. The `products` array is returned inside the `subscribe` method, where we set the `products` component property to the array emitted from the observable.

2. Modify the `ngOnInit` method so that it calls the newly created `getProducts` method:

```
ngOnInit(): void {  
    this.getProducts();  
}
```



We could have added the body of the `getProducts` method inside the `ngOnInit` method directly. We did not as component lifecycle event methods should be as clear and concise as possible. Always try to extract their logic in a separate method for clarity.

Run the application using the `ng serve` command, and you should see the product list displayed on the page successfully:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Figure 6.3: Product list

As depicted in the previous image, we have achieved the same result of displaying the product list as in *Chapter 5, Managing Complex Tasks with Services*, but using observables. It may not be evident at once, but we have set the foundation for working with the Angular HTTP client which is based on observables. In *Chapter 8, Communicating with Data Services over HTTP*, we will explore the HTTP client in more detail.

When we subscribe to observables, we are prone to potential memory leaks if we do not clean them up on time. In the following section, we will learn about different ways to accomplish that.

Unsubscribing from observables

When we subscribe to an observable, we create an observer that listens for changes in a data stream. The observer watches the stream continuously while the subscription remains active. When a subscription is active, it reserves memory in the browser and consumes certain resources. If we do not tell the observer to unsubscribe at some point and clean up any resources, the subscription to the observable will *possibly* lead to a memory leak.



An observer usually needs to unsubscribe when the Angular component that created the subscription must be destroyed.

Some of the most well-known techniques to use for unsubscribing from observables are the following:

- Unsubscribe from an observable manually
- Use the `async` pipe in a component template

Let's see both techniques in action in the following subsections.

Destroying a component

A component has lifecycle events we can use to hook on and perform custom logic, as we learned in *Chapter 3, Structuring User Interfaces with Components*. One of them is the `ngOnDestroy` event, which is called when the component is destroyed and no longer exists.

Recall `ProductListComponent` and `ProductViewComponent`, which we used earlier in our examples. They subscribe to the appropriate methods of `ProductsService` and `ProductViewService` upon component initialization. When components are destroyed, the reference of the subscriptions stays active, which may lead to unpredictable behavior. We need to manually unsubscribe when components are destroyed to clean up any resources properly:

1. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { Subscription } from 'rxjs';
```

2. Create the following property in the `ProductListComponent` class:

```
private productsSub: Subscription | undefined;
```

3. Assign the `productsSub` property to the subscription result in the `getProducts` method:

```
private getProducts() {  
    this.productsSub = this.productService.getProducts().  
        subscribe(products => {  
            this.products = products;  
        });  
}
```

4. Import the `OnDestroy` lifecycle hook from the `@angular/core` npm package:

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

5. Add `OnDestroy` to the implemented interface list of the `ProductListComponent` class:

```
export class ProductListComponent implements OnInit, OnDestroy
```

6. Implement the `ngOnDestroy` method as follows:

```
ngOnDestroy(): void {  
    this.productsSub?.unsubscribe();  
}
```

The `unsubscribe` method removes an observer from the active listeners of a subscription and cleans up any reserved resources.

That's a lot of boilerplate code to unsubscribe from a single subscription. It may quickly become unreadable and unmaintainable if we have many subscriptions.

Alternatively, we can use a particular type of operator called `takeUntilDestroyed`, which is available in the `@angular/core/rxjs-interop` package. We will explore the way of unsubscribing from observables using this operator in the product list component:

1. Open the `product-list.component.ts` file and import the `inject`, `DestroyRef`, and `takeUntilDestroyed` artifacts as follows:

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

```
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';
```

The `takeUntilDestroyed` artifact is an operator that unsubscribes from an observable when the component is destroyed.

2. Declare the following property to inject the `DestroyRef` service:

```
private destroyRef = inject(DestroyRef);
```

3. Modify the `getProducts` method as follows:

```
private getProducts() {  
    this.productService.getProducts().pipe(  
        takeUntilDestroyed(this.destroyRef)  
    ).subscribe(products => {  
        this.products = products;  
    });  
}
```

In the preceding method, we use the `pipe` operator to chain the `takeUntilDestroyed` operator with the subscription from the `getProducts` method of the `ProductsService` class. The `takeUntilDestroyed` operator accepts a parameter of the `DestroyRef` service.

4. Remove any code related to the `ngOnDestroy` method.

That's it! We have now converted our subscription to be more declarative and readable. However, the problem of maintainability still exists. Our components are now unsubscribing from their observables manually. We can solve that using a special-purpose Angular pipe, the `async` pipe, which allows us to unsubscribe automatically with less code.

Using the `async` pipe

The `async` pipe is a built-in Angular pipe used in conjunction with observables, and its role is two-fold. It helps us to type less code and saves us from having to set up and tear down a subscription. It automatically subscribes to an observable and unsubscribes when the component is destroyed. We will use it to simplify the code of the product list component:

1. Open the `product-list.component.ts` file and add the following `import` statements:

```
import { AsyncPipe } from '@angular/common';  
import { Observable } from 'rxjs';
```

2. Add the `AsyncPipe` class into the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-list',
  imports: [ProductDetailComponent, SortPipe, AsyncPipe],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

3. Convert the `products` component property to an observable:

```
products$: Observable<Product[]> | undefined;
```

4. Assign the `getProducts` method of the `ProductsService` class to the `products$` component property:

```
private getProducts() {
  this.products$ = this.productService.getProducts();
}
```

The body of the `getProducts` method has now been reduced to one line and has become more readable.

5. Open the `product-list.component.html` file and add the following snippet at the beginning of the file:

```
@let products = (products$ | async)!
```

In the preceding snippet, we subscribe to the `products$` observable using the `async` pipe and create a template variable using the `@let` keyword. The template variable has the same name as the respective component property we had previously, so we do not need to change the component template further.

That's it! We do not need to subscribe or unsubscribe from the observable manually anymore! The `async` pipe takes care of everything for us.

We have learned that observables react to application events and emit values asynchronously in registered observers. We could visualize observables as wrapper objects around emitted values. Angular enriches the reactivity field of web applications by providing a similar wrapper that works synchronously and reacts to application state changes.

Summary

It takes much more than a single chapter to cover in detail all the great things we can do with reactivity in Angular. The good news is that we have covered all the tools and classes we need for basic Angular development.

We learned what reactive programming is and how it can be used in Angular. We saw how to apply reactive techniques like observables to interact with data streams. We explored the RxJS library and how to use some operators to manipulate observables. We learned different ways of subscribing and unsubscribing from observables in Angular components.

The rest is just left to your imagination, so feel free to go the extra mile and put all of this knowledge into practice in your Angular applications. The possibilities are endless, and you have strategies ranging from promises and observables. You can leverage the incredible functionalities of the reactive patterns and build amazing reactive experiences for your Angular applications.

As we have already highlighted, the sky's the limit. However, we still have a long and exciting road ahead. In the next chapter, we will explore signals, an alternate reactive pattern built into the Angular framework. We will learn how to use Angular signals to handle the state of an Angular application.

7

Tracking Application State with Signals

Angular empowers developers to use built-in reactivity in their applications using **signals**. Angular signals are a synchronous approach to reactive programming that efficiently improves application performance and manages application state.

We met signals in previous chapters where we used the `input` method to exchange data between components and the `viewChild` method to query child components. The Signals API can be used in different parts of an Angular application, thus, its usage is scattered throughout the chapters of this book.

In this chapter, we will cover the following topics:

- Understanding signals
- Reading and writing signals
- Computed signals
- Cooperating with RxJS

Technical requirements

The chapter contains various code samples to walk you through the concept of Angular signals. You can find the related source code in the `ch07` folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Understanding signals

As we learned in *Chapter 3, Structuring User Interfaces with Components*, `Zone.js` plays a significant role in the performance of an Angular application. It triggers the Angular change detection mechanism when particular events occur inside the application. The framework checks every application component in each detection cycle and evaluates its bindings, degrading application performance.

The rationale of change detection with `Zone.js` is based on the fact that Angular cannot know when or where a change has happened inside the application. Inevitably, Angular developers try to limit change detection cycles using the following techniques:

- Configuring components with the `OnPush` change detection strategy
- Interacting manually with the change detection mechanism using the `ChangeDetectorRef` service

Signals improve how developers interact with the Angular change detection mechanism by simplifying and enhancing the preceding techniques according to application needs.

Angular signals provide more robust and ergonomic management of the change detection cycle based on reactivity. They watch how the application state changes and allow the framework to react by triggering change detection only in parts affected by the change.



Signals are an innovative feature of the Angular framework that will enable further improvements in the application's performance by introducing **zone-less applications** and **signal-based components** in the future.

Signals also act as containers for values, which the change detection mechanism must check. When a value changes, signals notify the framework about that change. The framework is responsible for triggering change detection and updating any signal consumers. A signal value can change either directly using **writable** signals or indirectly using **read-only** or **computed** signals.

In the following section, we will learn how writable signals work.

Reading and writing signals

A writable signal is indicated by the `signal` type from the `@angular/core` npm package.



You will need the source code of the Angular application we created in *Chapter 6, Reactive Patterns in Angular*, to follow along with the rest of the chapter. After you get the code, we suggest you remove the `key-logger` folder for simplicity.

Let's get started and learn how we can write a value in a signal:

1. Open the `app.component.ts` file and import the `signal` artifact from the `@angular/core` npm package:

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

2. Declare the following property in the `AppComponent` class as a `signal` and initialize it:

```
currentDate = signal(new Date());
```

3. Replace the `timestamp` variable in the `setTitle` property with the following snippet:

```
this.currentDate.set(new Date());
```

In the preceding snippet, we use the `set` method to write a new value in the signal. The method notifies the Angular framework that the value has changed, and it must run the change detection mechanism.

4. Modify the `title` property to use the value of the `currentDate` signal:

```
this.title = `${this.settings.title} (${this.currentDate()})`;
```

In the preceding snippet, we call the `currentDate` getter method to read the value of the signal.

Signals are a great choice in cases where the speed and performance of an application matters, such as:

- A dashboard page with widgets and live data that must be updated regularly, such as a stock exchange application.
- A component that needs to display properties from a large or complex object, such as the following:

```
const order = {
  no: '1',
  date: new Date(),
  products: [
    {
      id: 1,
      title: 'Keyboard',
      price: 100
    },
  ],
};
```

```
{
  id: 2,
  title: 'Microphone',
  price: 35
}
],
customerCode: '0002',
isCompleted: false
};
```

In this case, we can extract the object properties we want in a signal without involving the whole object in the change detection cycle, such as:

```
const orderDetails = signal({
  no: '1',
  customerCode: '0002',
  isCompleted: false
});
```

A similar method of signals that also triggers change detection is the update method. It is used when we want to set a new value on a signal based on its current value:

```
this.currentDate.update(d => {
  return new Date(d.getFullYear(), d.getMonth(), d.getDate(), 0, 0);
});
```

The preceding snippet will get the value of the currentDate signal in the d variable and use it to return a new Date object.

In the following section, we will explore how computed signals behave in an Angular application.

Computed signals

A computed or read-only signal depends on other signals, writable or computed. The value of a computed signal cannot change directly using the set or the update method, it can only change indirectly when the value of any of the other signals changes.

Let's see how it works:

1. Open the app.component.ts file and import the computed and Signal artifacts from the @angular/core npm package:

```
import {
  Component,
  inject,
  Signal,
  computed,
  signal
} from '@angular/core';
```

2. Change the type of the title component property to Signal:

```
title: Signal<string> = signal('');
```

The Signal type indicates that the signal is a computed one.

3. Remove the title assignment from the setTitle method and add it inside the constructor as follows:

```
constructor() {
  this.title$.subscribe(this.setTitle);
  this.title = computed(() => {
    return `${this.settings.title} (${this.currentDate()})`;
  });
}
```

In the preceding snippet, we use the computed function to set the value of the title signal. The value of the title signal depends on the currentDate signal. It is updated every 2 seconds when the value of the currentDate signal changes.

4. Open the app.component.html file and modify the <header> HTML element as follows:

```
<header>{{ title() }}</header>
```

5. Run the application using `ng serve` and verify that the title is updated correctly.

Computed signals have great performance when it comes to more complicated calculations than the preceding one due to the following reasons:

- The computed function executes when the signal value is first read on the template
- A new signal value is calculated only when the derived signals change
- Computed signals use a cache mechanism to memoize values and return them without recalculating

Although signals are a modern reactive approach for Angular, they are relatively new to the Angular ecosystem compared to RxJS. In the following section, we will learn how they can cooperate with RxJS in an Angular application.

Cooperating with RxJS

Signals and RxJS empower Angular applications with reactive capabilities. These libraries can complement each other to provide reactivity while using the benefits of the Angular framework. Signals was not built to replace RxJS but to provide an alternate reactive approach to developers with the following additional characteristics:

- Fine-grained reactivity
- Imperative programming
- Improved usage of the change detection mechanism

However, there are core parts in the Angular framework that still use RxJS and observables, such as the HTTP client and the router. Additionally, many developers prefer the declarative approach that the RxJS library provides out of the box.



At the time of writing, the Angular team is currently investigating and experimenting to make RxJS *optional* for Angular applications in the foreseeable future. They are also working to convert built-in APIs such as the HTTP client and router into signals.

Angular Signals provides a built-in API to cooperate with RxJS and observables. The signals API provides a function that can convert an observable into a signal:

1. Open the `product-list.component.ts` file and import the `inject` and `toSignal` artifacts:

```
import { Component, inject } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
```

The `@angular/core/rxjs-interop` npm package includes all the utility methods for handling signal and observable cooperation. The `toSignal` function can convert an observable into a signal.



The `rxjs-interop` package also contains utility methods for converting a signal to an observable. You can read more in *Reactive Patterns with RxJS and Angular Signals* by Lamis Chebbi (Packt Publishing).

2. Create the following signal in the ProductListComponent class:

```
products = toSignal(inject(ProductsService).getProducts(), {  
    initialValue: []  
});
```

We pass two parameters in the `toSignal` function: the observable we want to convert and an initial value optionally. In this case, we pass the `getProducts` method of the `ProductsService` class that returns an observable, and we also set the initial value of the signal to an empty array.

3. Open the `product-list.component.html` file and modify its contents as follows:

```
@if (products().length > 0) {  
    <h1>Products {{products().length}}</h1>  
}  
  
<ul class="pill-group">  
    @for (product of products() | sort; track product.id) {  
        <li class="pill" (click)="selectedProduct = product">  
            @switch (product.title) {  
                @case ('Keyboard') {  }  
                @case ('Microphone') {  }  
                @default {  }  
            }  
            {{product.title}}  
        </li>  
    } @empty {  
        <p>No products found!</p>  
    }  
</ul>  
  
<app-product-detail  
    [product]="selectedProduct"  
    (added)="onAdded()"></app-product-detail>
```

In the preceding template, we removed the top `@if` block and converted the `products` property into a signal. We do not need the `async` pipe because signals subscribe automatically to an observable.

4. To further clean up our component, we can remove any code that is related to the async pipe and observables since it is no longer needed. The resulting product-list.component.ts file should be the following:

```
import { Component, inject } from '@angular/core';
import { toSignal } from '@angular/core/rxjs-interop';
import { Product } from '../product';
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
import { SortPipe } from '../sort.pipe';
import { ProductsService } from '../products.service';

@Component({
  selector: 'app-product-list',
  imports: [ProductDetailComponent, SortPipe],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
  selectedProduct: Product | undefined;
  products = toSignal(inject(ProductsService).getProducts(), {
    initialValue: []
  });

  onAdded() {
    alert(` ${this.selectedProduct?.title} added to the cart!`);
  }
}
```

5. Run the application using `ng serve` and observe that the application output displays the product list.

The preceding snippet looks much simpler. Angular signals improve the developer experience and ergonomics in addition to the performance of our applications.

Summary

In this chapter, we explored signals, which is a new reactive pattern in Angular that is used for managing application state. We learned their rationale and how they compare with Zone.js. We explored examples of how to read and write values into signals. We also learned how to create computed signals that depend on values from other signals.

In the next chapter, we will learn how to use the Angular HTTP client and consume data from a remote endpoint.

8

Communicating with Data Services over HTTP

A real-world scenario for enterprise Angular applications is to connect to remote services and APIs to exchange data. The Angular HTTP client provides out-of-the-box support for communicating with services over HTTP. The interaction of an Angular application with the HTTP client is based on RxJS observable streams, giving developers a rich set of capabilities for data access.

There are many ways to connect to APIs through HTTP. In this book, we will only scratch the surface. Still, the insights covered in this chapter will give you all you need to connect your Angular applications to HTTP services in no time, leaving all you can do with them up to your creativity.

In this chapter, we will explore the following concepts:

- Communicating data over HTTP
- Introducing the Angular HTTP client
- Setting up a backend API
- Handling CRUD data in Angular
- Authentication and authorization with HTTP

Technical requirements

The chapter contains various code samples to walk you through the concept of the Angular HTTP client. You can find the related source code in the `ch08` folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Communicating data over HTTP

Before we dive into describing the Angular HTTP client and how to use it to communicate with servers, let's talk about native HTTP implementations first. Currently, if we want to communicate with a server over HTTP using JavaScript, we can use the JavaScript-native `fetch` API. It contains all the necessary methods to connect with a server and exchange data.

You can see an example of how to fetch data in the following code:

```
fetch(url)
  .then(response => {
    return response.ok ? response.text() : '';
  })
  .then(result => {
    if (result) {
      console.log(result);
    } else {
      console.error('An error has occurred');
    }
  });
});
```

Although the `fetch` API is promise-based, the promise it returns is not rejected if there is an error. Instead, the request is unsuccessful when the `ok` property is not in the `response` object.

If the request to the remote URL is completed, we can use the `text()` method of the `response` object to return the response text inside a new promise. Finally, in the second `then` callback, we display either the response text or a specific error message to the browser console.



To learn more about the `fetch` API, check out the official documentation at <https://developer.mozilla.org/docs/Web/API/fetch>.

We have already learned that observables are flexible for managing asynchronous operations. You are probably wondering how we can apply this pattern when consuming information from an HTTP service. So far, you will be becoming used to submitting asynchronous requests to AJAX services and then passing the response to a callback or a promise. Now, we will handle the call by returning an observable. The observable will emit the server response as an event in the context of a stream, which can be funneled through RxJS operators to better digest the response.

Let's convert the previous example with the `fetch` API to an observable. We use the `Observable` class to wrap the `fetch` call in an observable stream and replace the `console` methods with the appropriate observer object methods:

```
const request$ = new Observable(observer => {
  fetch(url)
    .then(response => {
      return response.ok ? response.text() : '';
    })
    .then(result => {
      if (result) {
        observer.next(result);
        observer.complete();
      } else {
        observer.error('An error has occurred');
      }
    });
});
```

In the preceding snippet, we use the following observer methods:

- `next`: This returns the response data to subscribers when they arrive
- `complete`: This notifies subscribers that no other data will be available in the stream
- `error`: This alerts subscribers that an error has occurred

That's it! We have now built a custom HTTP client. Of course, this isn't much. Our custom HTTP client only handles a `GET` operation to get data from a remote endpoint. We are not handling many other operations of the HTTP protocol, such as `POST`, `PUT`, and `DELETE`. It was, however, essential to realize all the heavy lifting the HTTP client in Angular is doing for us. Another important lesson is how easy it is to turn an asynchronous API into an observable API that fits nicely with the rest of our asynchronous concepts. So, let's continue with Angular's implementation of an HTTP service.

Introducing the Angular HTTP client

The HTTP client of the Angular framework is a separate Angular library that resides in the `@angular/common` npm package under the `http` namespace. The Angular CLI installs this package by default when creating a new Angular project.



You will need the source code of the Angular application we created in *Chapter 6, Reactive Patterns in Angular*, to follow along with the rest of the chapter. After you get the code, we suggest you remove the key-logger folder for simplicity.

To start using the Angular HTTP client, we need to import the `provideHttpClient` method in the `app.config.ts` file:

```
import { provideHttpClient } from '@angular/common/http';
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient()
  ]
};
```



Suppose we want to use the HTTP client in applications built with older versions of Angular. In that case, we need to import an Angular module, called `HttpClientModule`, from the `@angular/common/http` namespace into one of the modules of our application.

The `provideHttpClient` method exposes various Angular services we can use to handle asynchronous HTTP communication. The most basic is the `HttpClient` service, which provides a robust API and abstracts all operations required to handle asynchronous connections through the following HTTP methods:

- `get`: This performs a **GET** operation to fetch data
- `post`: This performs a **POST** operation to add new data
- `put/patch`: This performs a **PUT/PATCH** operation to update existing data
- `delete`: This performs a **DELETE** operation to remove existing data

The previous HTTP methods constitute the primary operations for **Create, Read, Update, Delete (CRUD)** applications. All the earlier methods of the Angular HTTP client return an observable data stream. Angular components can use the RxJS library to subscribe to those methods and interact with a remote API.



The Angular team is currently investigating and experimenting to see if they can make the use of RxJS optional in the framework. In that case, we might see an HTTP implementation that is based on signals. For the rest of this chapter, we will stick with observables because the Angular HTTP client does not support signals out of the box.

In the following section, we will explore how to use these methods and communicate with a remote API.

Setting up a backend API

A web CRUD application usually connects to a server and uses an HTTP backend API to perform operations on data. It fetches existing data, updates it, creates new data, or deletes it.

In a real-world scenario, you will most likely interact with a real backend API service through HTTP. In this book, we will use a fake API called **Fake Store API**.



The official Fake Store API documentation can be found at <https://fakestoreapi.com>.

The Fake Store API is a backend REST API available online that you can use when you need fake data for an e-commerce or e-shop web application. It can manage products, shopping carts, and users available in the JSON format. It exposes the following main endpoints:

- **products:** This manages a set of product items
- **cart:** This manages the shopping cart of a user
- **user:** This manages a collection of application users
- **login:** This handles user authentication



In this chapter, we will work only with the products and login endpoints. However, we will revisit the cart endpoint later in the book.

All operations that modify data do not persist them physically in a database. However, they return an indication of whether the operation was successful. All operations that get data return a predefined collection of items.

Handling CRUD data in Angular

CRUD applications are widely used in the Angular world. You will hardly find any web application that does not follow this pattern. Angular does a great job of supporting this type of application by providing the `HttpClient` service. In this section, we will explore the Angular HTTP client by interacting with the products endpoint of the Fake Store API.

Fetching data through HTTP

The `ProductListComponent` class uses the `ProductsService` class to fetch and display product data. Data is currently hardcoded into the `products` property of the `ProductsService` class. In this section, we will modify our Angular application to work with live data from the Fake Store API:

1. Open the `app.component.ts` file and remove the `providers` property from the `@Component` decorator. We will provide `APP_SETTINGS` directly through the application configuration file.
2. At this point, we can also remove the `title` property, the `title$` observable, the `setTitle` property, and the `constructor` of the component class:

```
export class AppComponent {  
    settings = inject(APP_SETTINGS);  
}
```

3. Open the `app.component.html` file and modify the `<header>` HTML element so that it uses the `settings` object directly:

```
<header>{{ settings.title }}</header>
```

4. Open the `app.config.ts` file and add the `APP_SETTINGS` provider as follows:

```
import { provideHttpClient } from '@angular/common/http';  
import { ApplicationConfig, provideZoneChangeDetection } from '@  
angular/core';  
import { provideRouter } from '@angular/router';  
  
import { routes } from './app.routes';
```

```
import { APP_SETTINGS, appSettings } from './app.settings';

export const AppConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
    { provide: APP_SETTINGS, useValue: appSettings }
  ]
};
```

We provide APP_SETTINGS from the application configuration file because we want it to be accessible globally in the application.

5. Open the `app.settings.ts` file and add a new property in the `AppSettings` interface that represents the URL of the Fake Store API:

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

export interface AppSettings {
  title: string;
  version: string;
  apiUrl: string;
}

export const appSettings: AppSettings = {
  title: 'My e-shop',
  version: '1.0',
  apiUrl: 'https://fakestoreapi.com'
};

export const APP_SETTINGS = new InjectionToken<AppSettings>('app.settings');
```



The URL of a backend API can also be added in environment files, as we will learn in *Chapter 14, Bringing Applications to Production*.

6. Open the `products.service.ts` file and modify the `import` statements accordingly:

```
import { HttpClient } from '@angular/common/http';
import { Injectable, inject } from '@angular/core';
import { Product } from './product';
import { Observable, of } from 'rxjs';
import { APP_SETTINGS } from './app.settings';
```

7. Create the following property in the `ProductsService` class that represents the API products endpoint:

```
private productsUrl = inject(APP_SETTINGS).apiUrl + '/products';
```

8. Modify the constructor to inject the `HttpClient` service:

```
constructor(private http: HttpClient) { }
```

9. Modify the `getProducts` method so that it uses the `HttpClient` service to get the list of products:

```
getProducts(): Observable<Product[]> {
  return this.http.get<Product[]>(this.productsUrl);
}
```

In the preceding method, we use the `get` method of the `HttpClient` class and pass the products endpoint of the API as a parameter. We also define the `Product` as a generic type in the `get` method to indicate that the response from the API contains a list of `Product` objects.

10. Convert the `products` property to an empty array:

```
private products: Product[] = [];
```

We will use this for local cache purposes later, in the *Modifying data through HTTP* section.

11. Open the `product-list.component.html` file and modify the `@if` block so that it checks if the `products` template variable exists:

```
@if (products) {
  <h1>Products {{products.length}}</h1>
}
```

We need to check if the variable exists because data is now fetched from the Fake Store API and there will be a network delay before the variable has a value.

If we run the application using the `ng serve` command, we should see an extended list of products from the API similar to the following:

Products (20)

- ◆ Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin
- ◆ BIYLACLESEN Women's 3-in-1 Snowboard Jacket Winter Coats
- ◆ DANVOUY Womens T Shirt Casual Cotton Short
- ◆ Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
- ◆ John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
- ◆ Lock and Love Women's Removable Hooded Faux Leather Moto Biker Jacket
- ◆ MBJ Women's Solid Short Sleeve Boat Neck V
- ◆ Mens Casual Premium Slim Fit T-Shirts
- ◆ Mens Casual Slim Fit
- ◆ Mens Cotton Jacket
- ◆ Opna Women's Short Sleeve Moisture
- ◆ Pierced Owl Rose Gold Plated Stainless Steel Double
- ◆ Rain Jacket Women Windbreaker Striped Climbing Raincoats
- ◆ Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED
- ◆ SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s
- ◆ Silicon Power 256GB SSD 3D NAND A55 SLC Cache Performance Boost SATA III 2.5
- ◆ Solid Gold Petite Micropave

Figure 8.1: Product list from the Fake Store API

The products endpoint supports passing a request parameter to limit the results returned from the API. As indicated at <https://fakestoreapi.com/docs#p-limit>, we can use a query parameter named **limit** to accomplish that task. Let's see how we can pass query parameters in the Angular HTTP client:

1. Open the `products.service.ts` file and import the `HttpParams` class from the `@angular/common/http` namespace:

```
import { HttpClient, HttpParams } from '@angular/common/http';
```

The `HttpParams` class is used to pass query parameters in an HTTP request.

2. Create the following variable inside the `getProducts` method:

```
const options = new HttpParams().set('limit', 10);
```



The `HttpParams` class is immutable. The following would not work because every operation returns a new instance:

```
const options = new HttpParams();
options.set('limit', 10);
```

The `set` method of the `HttpParams` class creates a new query parameter. If we wanted to pass additional parameters, we should chain more `set` methods, such as:

```
const options = new HttpParams()
.set('limit', 10)
.set('page', 1);
```

3. We use the second parameter of the `get` method to pass query parameters using the `params` property:

```
return this.http.get<Product[]>(this.productsUrl, {
  params: options
});
```

4. Save your changes, wait for the application to reload, and observe the application's output:

Products (10)

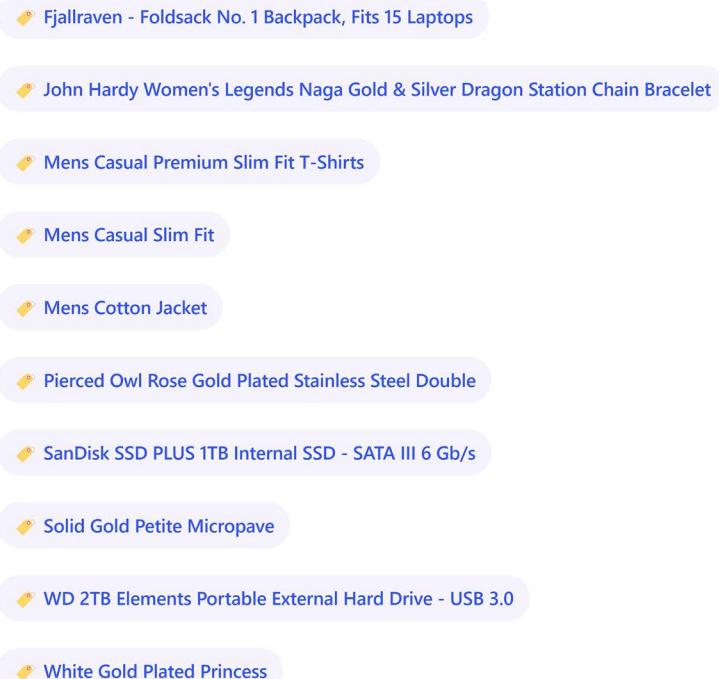


Figure 8.2: Product list

In the preceding list, all products are displayed with the same tag icon, which is the default one according to the @switch block in the `product-list.component.html` file:

```
<li class="pill" (click)="selectedProduct = product">
  @switch (product.title) {
    @case ('Keyboard') { 🖲️ }
    @case ('Microphone') { 🎤 }
    @default { ⚡ }
  }
  {{product.title}}
</li>
```

The @switch block relies on the product title property. We will change it so that it is based on the category property, which comes from the products endpoint of the API.

5. Open the `product.ts` file and replace the `categories` property with the following property:

```
category: string;
```

6. Open the `product-list.component.html` file and modify the @switch block as follows:

```
@switch (product.category) {  
    @case ('electronics') { 🖥 }  
    @case ('jewelery') { 💎 }  
    @default { 🎉 }  
}
```

7. We also need to modify the `product-detail.component.html` file because we replaced the `categories` property in step 1:

```
@if (product()) {  
    <p>You selected:  
        <strong>{{product()!.title}}</strong>  
    </p>  
    <p>{{product()!.price | currency:'EUR'}}</p>  
    <div class="pill-group">  
        <p class="pill">{{ product()!.category }}</p>  
    </div>  
    <button (click)="addToCart()">Add to cart</button>  
}
```

8. Save your changes, wait for the application to reload, and observe the application's output:

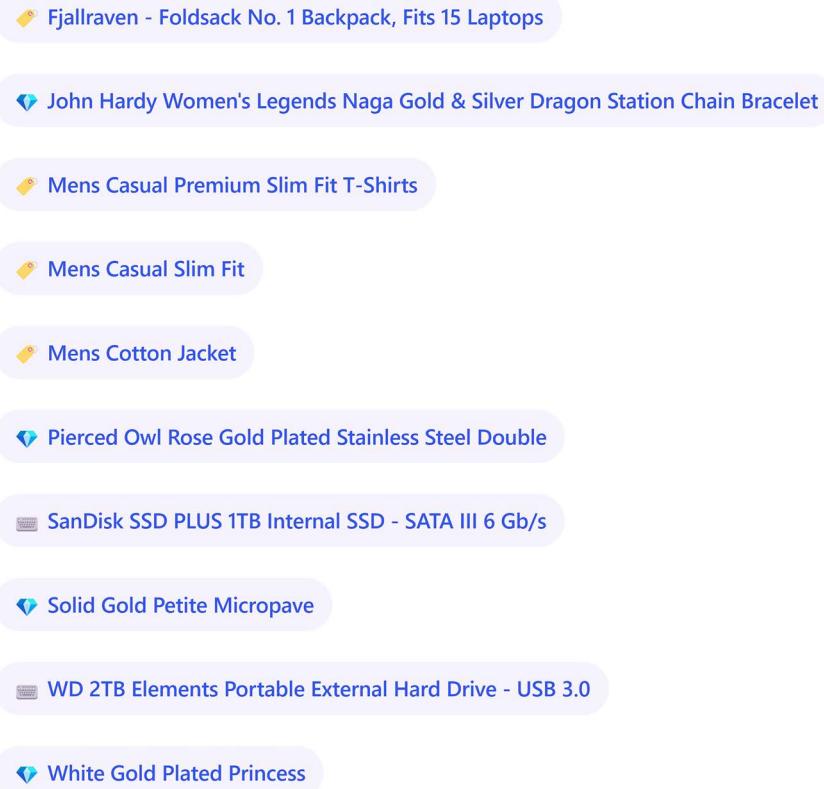


Figure 8.3: Product list with categories

If you click on a product in the list, you will notice that the product details are shown correctly:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€109.00

electronics

Add to cart

Figure 8.4: Product details

The product details component continues to work as expected because we pass the selected product as an input property from the product list:

```
<app-product-detail  
  [product]="selectedProduct"  
  (added)="onAdded()"  
></app-product-detail>
```

We will change the previous behavior and get the product details directly from the API using an HTTP GET request. The Fake Store API contains an endpoint method that we can use to get the details for a specific product based on its ID:

1. Open the `products.service.ts` file and create a new `getProduct` method that accepts the product id as a parameter and initiates a GET request to the API based on that id:

```
getProduct(id: number): Observable<Product> {  
  return this.http.get<Product>(`${this.productsUrl}/${id}`);  
}
```

The preceding method uses the `get` method of the `HttpClient` service. It accepts the products endpoint URL followed by the product id as a parameter.

2. Open the `product-detail.component.ts` file and modify the `import` statements as follows:

```
import { CommonModule } from '@angular/common';  
import {  
  Component,  
  input,  
  output,  
  OnChanges  
} from '@angular/core';  
import { Product } from '../product';  
import { Observable } from 'rxjs';  
import { ProductsService } from '../products.service';
```

3. Add the following property in the `ProductDetailComponent` class:

```
id = input<number>();
```

The `id` component property will be used to pass the ID of the selected product from the list.

4. Replace the product input property with the following observable:

```
product$: Observable<Product> | undefined;
```

The product\$ property will be used to call the getProduct method from the service.

5. Add a constructor in the ProductDetailComponent class and inject ProductsService:

```
constructor(private productService: ProductsService) { }
```

6. Add OnChanges in the list of implemented interfaces:

```
export class ProductDetailComponent implements OnChanges
```

7. Implement the ngOnChanges method as follows:

```
ngOnChanges(): void {
  this.product$ = this.productService.getProduct(this.id());
}
```

In the preceding method, we assign the value of the getProduct method from ProductsService to the product\$ component property every time a new id is passed using the input binding.

8. Open the product-detail.component.html file and modify its content so that it uses the product\$ observable:

```
@let product = (product$ | async);
@if (product) {
  <p>You selected:
    <strong>{{product.title}}</strong>
  </p>
  <p>{{product.price | currency:'EUR'}}</p>
  <div class="pill-group">
    <p class="pill">{{ product.category }}</p>
  </div>
  <button (click)="addToCart()">Add to cart</button>
}
```

9. Finally, open the product-list.component.html file and bind the id of the selectedProduct property to the id input binding of the <app-product-detail> component:

```
<app-product-detail
  [id]="selectedProduct?.id"
```

```
(added)="onAdded()"></app-product-detail>
```

If we run the application using the `ng serve` command and select a product from the list, we will verify that the product detail is displayed correctly.

We have learned how to get a list of items and a single item from a backend API and covered the **Read** part of a CRUD operation. In the following section, we cover the remaining parts of a CRUD operation, which are mainly concerned with modifying data.

Modifying data through HTTP

Modifying data in a CRUD application usually refers to adding new data and updating or deleting existing data. To demonstrate how to implement such functionality in an Angular application using the HTTP client, we will make the following changes to our application:

- Create an Angular component to add new products
- Modify the product detail component to change the price of an existing product
- Add a button in the product detail component to delete an existing product

We have already mentioned that no HTTP operations persist data physically in the Fake Store API, so we need to implement a local cache mechanism for our product data and interact with it directly in the products service:

1. Open the `products.service.ts` file and import the `map` RxJS operator:

```
import { Observable, map, of } from 'rxjs';
```

2. Modify the `getProducts` method as follows:

```
getProducts(): Observable<Product[]> {
  const options = new HttpParams().set('limit', 10);
  return this.http.get<Product[]>(this.productsUrl, {
    params: options
  }).pipe(map(products => {
    this.products = products;
    return products;
  }));
}
```

The preceding method fills the `products` array with data from the API and returns product data as an observable.

3. Modify the `getProduct` method so that it uses the `products` array to return a product object instead of the Fake Store API:

```
getProduct(id: number): Observable<Product> {
  const product = this.products.find(p => p.id === id);
  return of(product!);
}
```

We now have our products service in place and can start building the component for adding new products.

Adding new products

To add a new product through our application, we need to send its details to the Fake Store API:

1. Open the `products.service.ts` file and add the following method:

```
addProduct(newProduct: Partial<Product>): Observable<Product> {
  return this.http.post<Product>(this.productsUrl, newProduct).pipe(
    map(product => {
      this.products.push(product);
      return product;
    })
  );
}
```

In the preceding snippet, we use the `post` method of the `HttpClient` class and pass the `products` endpoint of the API along with a new product object as parameters.



We define the new product as `Partial` because new products do not have an ID.

The generic type defined in the `post` method indicates that the returned product from the API is a `Product` object. We also add the new product into the local cache and return it.

2. Run the following Angular CLI command to create a new component:

```
ng generate component product-create
```

3. Open the `product-create.component.ts` file and add the following `import` statement:

```
import { ProductsService } from '../products.service';
```

4. Create a constructor and inject the ProductsService class:

```
constructor(private productsService: ProductsService) {}
```

5. Add the following method to the component class:

```
createProduct(title: string, price: string, category: string) {
  this.productsService.addProduct({
    title,
    price: Number(price),
    category
  }).subscribe();
}
```



We do not need to unsubscribe when interacting with the Angular HTTP client because the framework will do it automatically for us.

The preceding method accepts the product details as parameters and calls the addProduct method of the ProductsService class. We use the native Number function to convert the price value to a number because it will be passed as a string from the template.

6. Open the `product-create.component.html` file and replace its content with the following HTML template:

```
<h1>Add new product</h1>
<div>
  <label for="title">Title</label>
  <input id="title" #title />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" #price type="number" />
</div>
<div>
  <label for="category">Category</label>
  <select id="category" #category>
    <option>Select a category</option>
    <option value="electronics">Electronics</option>
  </select>
</div>
```

```
<option value="jewelery">Jewelery</option>
<option>Other</option>
</select>
</div>
<div>
  <button (click)="createProduct(title.value, price.value, category.
value)">Create</button>
</div>
```

In the preceding template, we bind the `createProduct` method to the `click` event of the `Create` button and pass the value of the `<input>` and `<select>` HTML elements using the respective template reference variables.

7. Open the `global.styles.css` file and add the following CSS style:

```
input {
  border-radius: 4px;
  padding: 8px;
  margin-bottom: 16px;
  border: 1px solid #BDBDBD;
}
```

Also, move the button-related styles from the `product-detail.component.css` file in the global CSS styles file.

8. Open the `product-create.component.css` file and add the following CSS styles to give a nice look and feel to our new component:

```
input {
  width: 200px;
}

select {
  border-radius: 4px;
  padding: 8px;
  margin-bottom: 16px;
  border: 1px solid #BDBDBD;
  width: 220px;
}
```

```
label {  
  margin-bottom: 4px;  
  display: block;  
}
```

9. Open the `product-list.component.ts` file and import the `ProductCreateComponent` class:

```
import { AsyncPipe } from '@angular/common';  
import { Component, OnInit } from '@angular/core';  
import { Observable } from 'rxjs';  
import { Product } from '../product';  
import { ProductDetailComponent } from '../product-detail/product-  
detail.component';  
import { SortPipe } from '../sort.pipe';  
import { ProductsService } from '../products.service';  
import { ProductCreateComponent } from '../product-create/product-  
create.component';  
  
@Component({  
  selector: 'app-product-list',  
  imports: [  
    ProductDetailComponent,  
    SortPipe,  
    AsyncPipe,  
    ProductCreateComponent  
  ],  
  templateUrl: './product-list.component.html',  
  styleUrls: ['./product-list.component.css'  
})
```

10. Finally, open the `product-list.component.html` file and add the following snippet at the end of the template:

```
<app-product-create></app-product-create>
```

If we now run our Angular application using the `ng serve` command, we should see the component for adding new products at the end of the page:

Add new product

Title

Price

Category

Select a category

Create

Figure 8.5: Create a product

To experiment, try to add a new product by filling in its details, clicking on the **Create** button, and verifying that the new product has been added to the list.

The next feature we will add to our application is to modify data by changing the price of an existing product.

Updating product price

The price of a product in an e-commerce application may need to change at some point. We need to provide a way for our users to update that price through our application:

1. Open the `products.service.ts` file and add a new method for updating a product:

```
updateProduct(id: number, price: number): Observable<Product> {
    return this.http.patch<Product>(`${this.productsUrl}/${id}`, {
        price
    }).pipe(
        map(product => {
            const index = this.products.findIndex(p => p.id === id);
            this.products[index].price = price;
            return product;
        })
    );
}
```

In the preceding method, we use the `patch` method of the `HttpClient` class to send the details of the product that we want to modify to the API. We also update the price of the selected product in the local cache of products and return it.



Alternatively, we could have used the `put` method of the HTTP client. The `patch` method should be used when we want to update only a subset of an object, whereas the `put` method interacts with all object properties. In this case, we do not want to update the product title, so we use the `patch` method. Both methods accept the API endpoint and the object we want to update as parameters.

2. Add the following method to the `ProductDetailComponent` class:

```
changePrice(product: Product, price: string) {
    this.productService.updateProduct(product.id, Number(price)).
    subscribe();
}
```

The preceding method accepts an existing product and its new price as parameters and calls the `updateProduct` method of the `ProductsService` class.

3. Open the `product-detail.component.html` file and add an `<input>` and a `<button>` element after the paragraph element of the price:

```
@let product = (product$ | async);
@if (product) {
<p>You selected:
<strong>{{product.title}}</strong>
</p>
<p>{{product.price | currency:'EUR'}}</p>
<input placeholder="New price" #price type="number" />
<button
    class="secondary"
    (click)="changePrice(product, price.value)">
    Change
</button>
<div class="pill-group">
    <p class="pill">{{ product.category }}</p>
```

```
</div>
<button (click)="addToCart()">Add to cart</button>
}
```

The `<input>` element is used to enter the new price of the product and defines the `price` template reference variable. The `click` event of the `<button>` element is bound to the `changePrice` method that passes the current product object and the value of the `price` variable.

- Finally, open the `product-detail.component.css` file and add the following CSS styles:

```
button.secondary {
  display: inline;
  margin-left: 5px;
  --button-accent: var(--vivid-pink);
}
```

- Run the `ng serve` command to start the Angular application and select a product from the list. The product details should look like the following:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€109.00

New price

Change

electronics

Add to cart

Figure 8.6: Product details

- Enter a price in the **New price** input box and click the **Change** button. The existing price should be updated to reflect the change, for example:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€79.00

Change

electronics

Add to cart

Figure 8.7: Product details with changed price

We can now modify a product by changing its price.



Remember that changes in products that come from the Fake Store API are not physically persisted. If you change the price and refresh the browser, it will restore the initial price.

The next and final step of our CRUD application will be to delete an existing product.

Removing a product

Deleting a product from an e-shop application is not very common. However, we need to provide functionality for it in case users enter incorrect or invalid data and want to delete it afterward. In our application, deleting an existing product will be done with the product details component:

1. Open the `products.service.ts` file and import the `tap` operator from the `rxjs` package:

```
import { Observable, map, of, tap } from 'rxjs';
```

2. Add the following method to the `ProductsService` class:

```
deleteProduct(id: number): Observable<void> {
  return this.http.delete<void>(`${this.productsUrl}/${id}`).pipe(
    tap(() => {
      const index = this.products.findIndex(p => p.id === id);
      this.products.splice(index, 1);
    })
  );
}
```

In the preceding method, we use the `delete` method of the `HttpClient` class, passing the products endpoint and the product id we want to delete in the API. We are also using the `splice` method of the `products` array to remove the product from the local cache.

The return type of the method is set to `Observable<void>` because we are not currently interested in the result of the HTTP request. We only need to know if it was successful or not. We also use the `tap` RxJS operator because we are not altering the returned value from the observable.

3. Open the `product-detail.component.ts` file and create a new output property in the `ProductDetailComponent` class:

```
deleted = output();
```

The preceding property will notify the `ProductListComponent` that the selected product has been deleted.

4. Create the following method, which calls the `deleteProduct` method of the `ProductsService` class and triggers the `deleted` output event:

```
remove(product: Product) {
  this.productService.deleteProduct(product.id).subscribe(() => {
    this.deleted.emit();
  });
}
```

5. Open the `product-detail.component.html` file, create a `<button>` element, and bind its `click` event to the `emit` method of the `deleted` output:

```
@let product = (product$ | async);
@if (product) {
  <p>You selected:
  <strong>{{product.title}}</strong>
  </p>
  <p>{{product.price | currency:'EUR'}}</p>
  <input placeholder="New price" #price type="number" />
  <button
    class="secondary"
    (click)="changePrice(product, price.value)">
    Change
  </button>
}
```

```
<div class="pill-group">
  <p class="pill">{{ product.category }}</p>
</div>
<div class="button-group">
  <button (click)="addToCart()">Add to cart</button>
  <button class="delete" (click)="remove(product)">Delete</button>
</div>
}
```

In the preceding snippet, we grouped the two buttons in a `<div>` HTML element so that they appear side by side.

6. Add an appropriate style for the new button and the button group in the `product-detail.component.css` file:

```
button.delete {
  display: inline;
  margin-left: 5px;
  --button-accent: var(--hot-red);
}

.button-group {
  display: flex;
  flex-direction: row;
  align-items: start;
  flex-wrap: wrap;
}
```

7. Open the `product-list.component.html` file and add a binding to the deleted event of the `<app-product-detail>` component:

```
<app-product-detail
  [id]="selectedProduct?.id"
  (added)="onAdded()"
  (deleted)="selectedProduct = undefined"
></app-product-detail>
```

If we run the application using the `ng serve` command and select a product from the list, we should see something like the following:

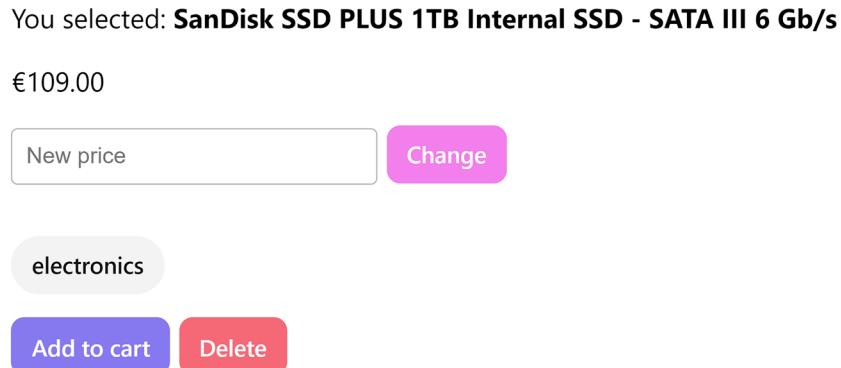
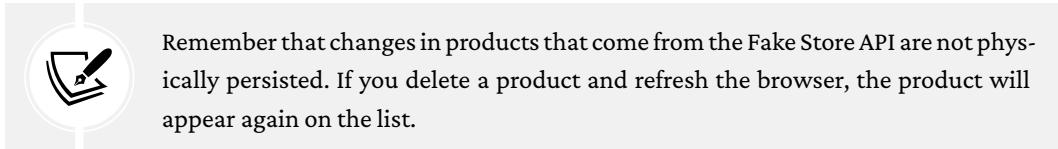
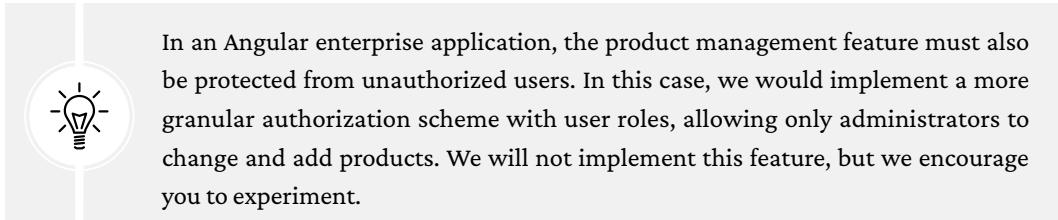


Figure 8.8: Product details

The product detail component now has a **Delete** button that deletes the product and removes it from the list when it is clicked.



The e-shop application we have built so far has an **Add to cart** button that we can use to add a product to a shopping cart. The button does not do much yet, but we will implement the full cart functionality in the following chapters. According to the documentation of the Fake Store API, shopping carts are only available to authenticated users, so we must ensure that the **Add to cart** button will only be available to them in our application.



In the following section, we will learn about authentication and authorization in Angular.

Authentication and authorization with HTTP

The Fake Store API provides an endpoint for authenticating users. It contains a login method that accepts a username and a password as parameters and returns an authentication token. We will use the authentication token in our application to differentiate between a logged-in user and a guest.



A predefined pool from the users endpoint at <https://fakestoreapi.com/users> provides the username and password.

We will explore the following authentication and authorization topics in this section:

- Authenticating with a backend API
- Authorizing users for certain features
- Authorizing HTTP requests using interceptors

Let's get started with the topic of authenticating with the Fake Store API.

Authenticating with a backend API

In Angular real-world applications, we usually create an Angular component, allowing users to log in and out of the application. An Angular service will communicate with the API and handle all authentication tasks.

Let's get started by creating the authentication service:

1. Run the following command to create a new Angular service:

```
ng generate service auth
```

2. Open the `auth.service.ts` file and modify the import statements as follows:

```
import { Injectable, computed, inject, signal } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable, tap } from 'rxjs';
import { APP_SETTINGS } from './app.settings';
```

3. Create the following properties in the AuthService class:

```
private accessToken = signal('');
private authUrl = inject(APP_SETTINGS).apiUrl + '/auth';
isLoggedIn = computed(() => this.accessToken() !== '');
```

In the preceding snippet, the `accessToken` signal will store the authentication token from the API, and the `isLoggedIn` signal indicates whether the user is logged in. The logged-in status of the user depends on whether the `accessToken` property has a value.



Signals can be used not only in Angular components but also inside services.

The `authUrl` property points to the authentication endpoint URL of the Fake Store API.

4. Inject the `HttpClient` class in the constructor:

```
constructor(private http: HttpClient) { }
```

5. Create a `login` method to allow users to log in to the Fake Store API:

```
login(username: string, password: string): Observable<string> {
  return this.http.post<string>(this.authUrl + '/login', {
    username, password
  }).pipe(tap(token => this.accessToken.set(token)));
}
```

The preceding method initiates a POST request to the API, using the `login` endpoint and passing `username` and `password` in the request body. The observable returned from the POST request is passed to the `tap` operator, which updates the `accessToken` signal.

6. Create a `logout` method that resets the `accessToken` signal:

```
logout() {
  this.accessToken.set('');
}
```

We have already set up the business logic for authenticating users in our Angular application. In the following section, we will learn how to use it to control authorization in the application.

Authorizing user access

First, we will create an authentication component that will allow our users to log in and out of the application:

1. Run the following command to create a new Angular component:

```
ng generate component auth
```

2. Open the auth.component.ts file and add the following import statement:

```
import { AuthService } from '../auth.service';
```

3. Inject AuthService in the component's constructor:

```
constructor(public authService: AuthService) {}
```

In the preceding snippet, we use the public access modifier to inject AuthService because we want it to be accessible from the component's template.

4. Create the following methods in the AuthComponent class:

```
login() {
  this.authService.login('david_r', '3478*#54').subscribe();
}

logout() {
  this.authService.logout();
}
```

In the preceding snippet, the login method uses predefined credentials from the users endpoint.

5. Open the auth.component.html file and replace its content with the following HTML template:

```
@if (!authService.isLoggedIn()) {
  <button (click)="login()">Login</button>
} @else {
  <button (click)="logout()">Logout</button>
}
```

The preceding template contains two <button> HTML elements for login/logout purposes. Each button is displayed conditionally according to the value of the isLoggedIn signal of the AuthService class.

We can now leverage the `isLoggedIn` signal in the product detail component and toggle the visibility of the **Add to cart** button:

1. Open the `product-detail.component.ts` file and add the following import statement:

```
import { AuthService } from './auth.service';
```

2. Inject `AuthService` in the constructor of the `ProductDetailComponent` class:

```
constructor(private productService: ProductsService, public authService: AuthService) { }
```

3. Open the `product-detail.component.html` file and use an `@if` block to display the **Add to cart** button conditionally:

```
@if (authService.isLoggedIn()) {  
  <button (click)="addCart()">Add to cart</button>  
}
```

4. Open the `app.component.ts` file and import the `AuthComponent` class:

```
import { Component, inject } from '@angular/core';  
import { RouterOutlet } from '@angular/router';  
import { ProductListComponent } from './product-list/product-list.component';  
import { CopyrightDirective } from './copyright.directive';  
import { APP_SETTINGS } from './app.settings';  
import { AuthComponent } from './auth/auth.component';  
  
@Component({  
  selector: 'app-root',  
  imports: [  
    RouterOutlet,  
    ProductListComponent,  
    CopyrightDirective,  
    AuthComponent  
  ],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'  
})
```

5. Open the `app.component.html` file and add the `<app-auth>` tag inside the `<header>` HTML element:

```
<header>
  {{ settings.title }}
  <app-auth></app-auth>
</header>
```

To try the authentication feature in the application, follow these steps:

1. Run the `ng serve` command to start the application and navigate to `http://localhost:4200`.
2. Select a product from the list and verify that the **Add to cart** button is not visible.
3. Click the **Login** button in the top-left corner of the page. The text should change to **Logout** after you have logged in successfully to the Fake Store API, and the **Add to cart** button should appear.

Congratulations! You have added basic authentication and authorization patterns to your Angular application.

It is common in enterprise applications to perform authorization in the business logic layer while communicating with the backend API. The backend API often requires certain method calls to pass the authentication token in each request through headers. We will learn how to work with **HTTP headers** in the following section.

Authorizing HTTP requests

The Fake Store API does not require authorization while communicating with its endpoints. However, suppose we are working with a backend API that expects all HTTP requests to contain an authentication token using HTTP headers. A common pattern in web applications is to include the token in an **Authorization** header. We can use HTTP headers in an Angular application by importing the `HttpHeaders` class from the `@angular/common/http` namespace and modifying our methods accordingly. Here is an example of how the `getProducts` method should look:

```
getProducts(): Observable<Product[]> {
  const options = {
    params: new HttpParams().set('limit', 10),
    headers: new HttpHeaders({ Authorization: 'myToken' })
  };
  return this.http.get<Product[]>(this.productsUrl, options).
    pipe(map(products => {
      this.products = products;
```

```
    return products;
  }));
}
```



For simplicity, we are using a hardcoded value for the authentication token. In a real-world scenario, we may get it from the local storage of the browser or some other means.

All `HttpClient` methods accept an optional object as a parameter for passing additional options to an HTTP request, including HTTP headers. To set headers, we use the `headers` property of the `options` object and create a new instance of the `HttpHeaders` class as a value. The `HttpHeaders` object is a key-value pair that defines custom HTTP headers.

Now, imagine what will happen if we need to pass the authentication token in all methods of the `ProductsService` class. We should go to each of them and write the same code repeatedly. Our code could quickly become cluttered and difficult to test. Luckily, the Angular HTTP client has another feature we can use to help us in such a situation called **interceptors**.

An HTTP interceptor is an Angular service that intercepts HTTP requests and responses that pass through the Angular HTTP client. It can be used in the following scenarios:

- When we want to pass custom HTTP headers in every request, such as an authentication token
- When we want to display a loading indicator while we wait for a response from the server
- When we want to provide a logging mechanism for every HTTP communication

In our case, we can create an interceptor for passing the authentication token to each HTTP request:

1. Run the following command to create a new interceptor:

```
ng generate interceptor auth
```

2. Open the `app.config.ts` file and import the `withInterceptors` function from the `@angular/common/http` namespace:

```
import { provideHttpClient, withInterceptors } from '@angular/
common/http';
```

The `withInterceptors` function is used to register an interceptor with the HTTP client.

3. Import the interceptor we created in the previous step using the following statement:

```
import { authInterceptor } from './auth.interceptor';
```

4. Modify the provideHttpClient method to register the authInterceptor:

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(withInterceptors([authInterceptor])),
    { provide: APP_SETTINGS, useValue: appSettings }
  ]
};
```

The withInterceptors function accepts a list of registered interceptors, and their order matters. In the following diagram, you can see how interceptors process HTTP requests and responses according to their order:

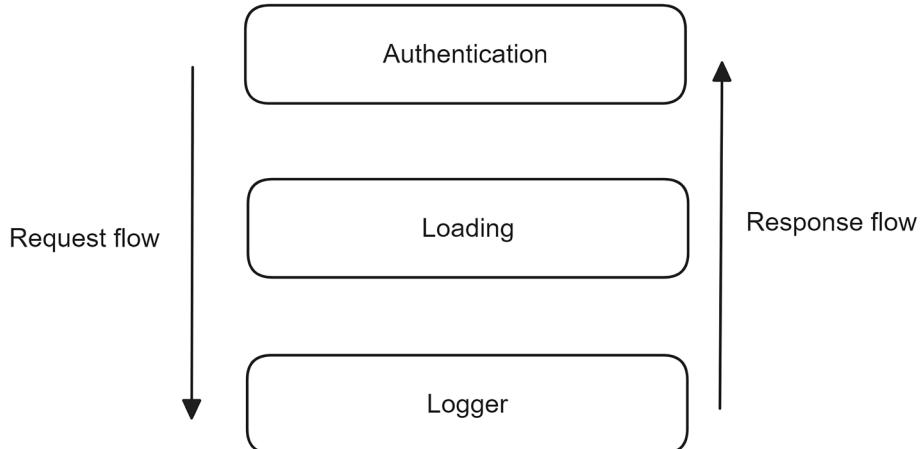


Figure 8.9: Execution order of Angular interceptors



By default, the last interceptor before sending the request to the server is a built-in Angular service named **HttpBackend**.

5. Open the auth.interceptor.ts file and modify the arrow function of the authInterceptor function as follows:

```
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const authReq = req.clone({
    setHeaders: { Authorization: 'myToken' }
  });
  next(authReq);
}
```

```
});  
return next(authReq);  
};
```

The arrow function accepts the following parameters: `req`, which indicates the current request, and `next`, which is the next interceptor in the chain. In the preceding snippet, we use the `clone` method to modify the existing request because HTTP requests are immutable by default. Similarly, due to the immutable nature of HTTP headers, we use the `setHeaders` method to update them. Finally, we delegate the request to the next interceptor using the `handle` method.

Interceptors can use the `inject` method to get dependencies that they may need from the Angular DI mechanism. For example, if we wanted to use the `AuthService` class inside the interceptor, we could modify it as follows:

```
import { inject } from '@angular/core';  
import { HttpInterceptorFn } from '@angular/common/http';  
import { AuthService } from './auth.service';  
  
export const authInterceptor: HttpInterceptorFn = (req, next) => {  
  const authService = inject(AuthService);  
  const authReq = req.clone({  
    setHeaders: { Authorization: 'myToken' }  
  });  
  return next(authReq);  
};
```

In applications built with older versions of the Angular framework, you may notice that interceptors are TypeScript classes instead of pure functions. To register an interceptor with the HTTP client, we need to add the following `provide` object literal in the `providers` array of the module, which also provides the `HttpClientModule`:



```
{  
  provide: HTTP_INTERCEPTORS,  
  useClass: AuthInterceptor,  
  multi: true  
}
```

In the preceding snippet, `HTTP_INTERCEPTORS` is an injection token that can be provided multiple times as indicated by the `multi` property.

Angular interceptors have many uses, and authorization is one of the most basic. Passing authentication tokens during HTTP requests is a common scenario in enterprise web applications.

Summary

Enterprise web applications must exchange information with a backend API almost daily. The Angular framework enables applications to communicate with an API over HTTP using the Angular HTTP client. In this chapter, we explored the essential parts of the Angular HTTP client.

We learned to move away from the traditional `fetch` API and use observables to communicate over HTTP. We explored the basic parts of a CRUD application using the Fake Store API as our backend. We investigated how to implement authentication and authorization in Angular applications. Finally, we learned what Angular interceptors are and how to use them to authorize HTTP calls.

Now that we know how to consume data from a backend API in our components, we can further improve the user experience of our application. In the next chapter, we will learn how to load our components through navigation using the Angular router.

9

Navigating through Applications with Routing

In previous chapters, we did a great job of separating concerns and adding different layers of abstraction to increase the maintainability of an Angular application. However, we have barely focused on the application's UX.

Our user interface is bloated, with components scattered across a single screen. We must provide a better navigational experience for users and a logical way to change the application's view intuitively. Now is the right time to incorporate routing and split the different areas of interest into pages, connected by a grid of links and URLs.

So, how do we deploy a navigation scheme between components of an Angular application? We use the Angular router and create custom links for our components to react to.

This chapter contains the following sections:

- Introducing the Angular router
- Configuring the main routes
- Organizing application routes
- Passing parameters to routes
- Enhancing navigation with advanced features

Technical requirements

The chapter contains various code samples to walk you through routing in the Angular framework. You can find the related source code in the ch09 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Introducing the Angular router

In traditional web applications, when we wanted to change from one view to another, we needed to request a new page from the server. The browser would create a URL for the view and send it to the server. The browser would then reload the page as soon as the client received a response. It was a process that resulted in round trip time delays and a bad user experience for our applications:

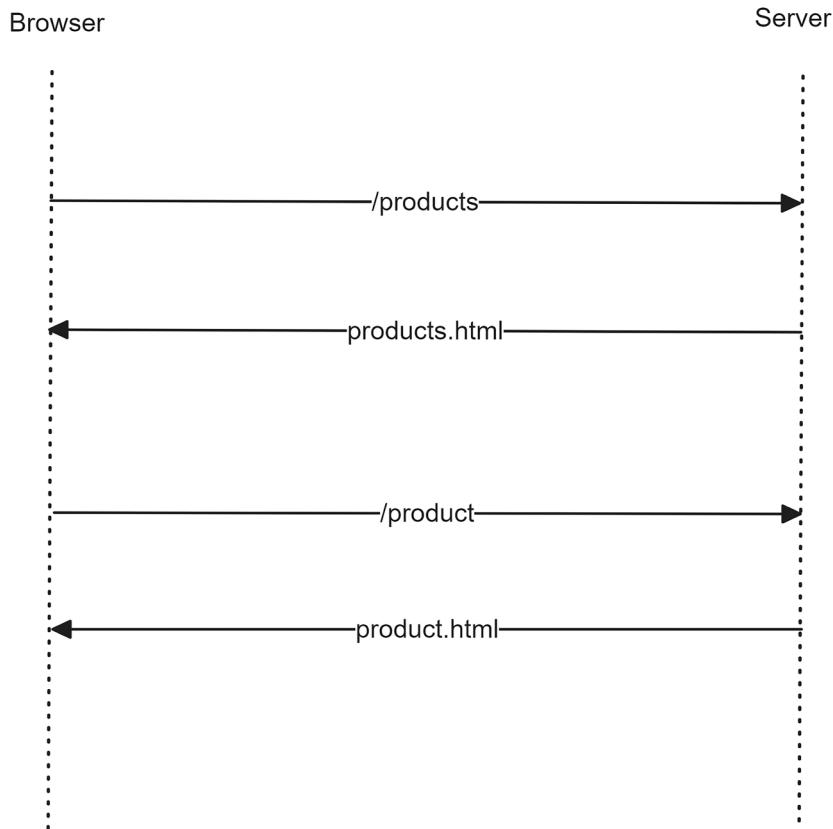


Figure 9.1: Routing in traditional web applications

Modern web applications using JavaScript frameworks such as Angular follow a different approach. They handle changes between views or components on the client side without bothering the server. They contact the server once during bootstrapping to get the main HTML file. The router on the client intercepts and handles any subsequent URL changes. These applications are called **Single-Page Applications (SPAs)** because they do not cause a full reload of a page:

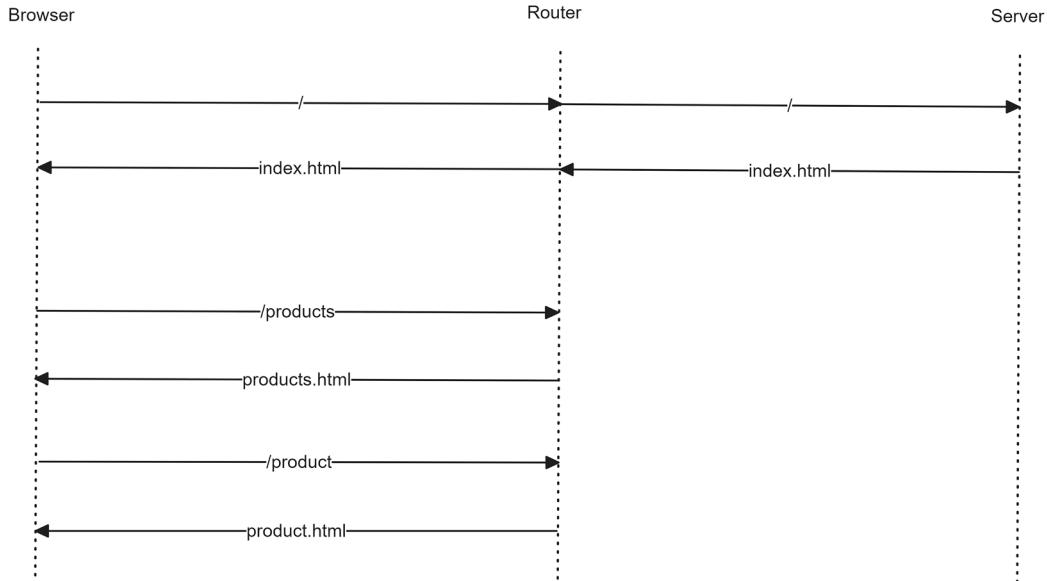


Figure 9.2: SPA architecture

The Angular framework provides the `@angular/router` npm package, which we can use to navigate between different components in an Angular application.

Adding routing in an Angular application involves the following steps:

1. Specifying the base path for the Angular application
2. Using an appropriate directive or service from the `@angular/router` npm package
3. Configuring different routes for the Angular application
4. Deciding where to render components upon navigation

In the following sections, we will learn the basics of Angular routing before diving deeper into hands-on examples.

Specifying a base path

As we have already seen, modern and traditional web applications react differently when a URL changes inside the application. The architecture of each browser plays an essential part in this behavior. Older browsers initiate a new request to the server when the URL changes. Modern browsers, also known as **evergreen** browsers, can change the URL and the browser history when navigating in different views without sending a request to the server, using a technique called `pushState`.



HTML5 `pushState` allows in-app navigation without causing a full reload of a page and is supported by all modern browsers.

An Angular application must set the `<base>` HTML tag in the `index.html` file to enable `pushState` routing:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The `href` attribute informs the browser of the path it should follow when loading application resources. The Angular CLI automatically adds the tag when creating a new application and sets the `href` value to the application root, `/`. If your application resides in a different folder from the root, you should name it after that folder.

Enabling routing in Angular applications

The Angular router is enabled by default in new Angular applications, as indicated by the `provideRouter` method in the `app.config.ts` file:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes)
  ]
};
```



In applications built with older versions of the Angular framework, the router is enabled by importing the `RouterModule` class in the main application module and using its `forRoot` method to define the routing configuration.

The `provideRouter` method enables us to use a set of Angular artifacts related to routing:

- Services to perform common routing tasks such as navigation
- Directives that we can use in our components to enrich them with navigation logic

It accepts a single parameter, which is the routing configuration of the application, and is defined by default in the `app.routes.ts` file.

Configuring the router

The `app.routes.ts` file contains a list of `Routes` objects that specify which routes exist in the application and which components should respond to a specific route. It looks like the following:

```
const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: '**', component: PageNotFoundComponent }
];
```



In applications built with older versions of the Angular framework, you may notice that the route configuration is defined in a dedicated `app-routing.module.ts` file.

Each route definition object contains a `path` property, which is the URL path of the route, and a `component` property that defines which component will be loaded when the application navigates to that path.



The value of a `path` property should not contain a leading `/`.

Navigation in an Angular application can occur manually by changing the browser URL or navigating using in-app links. The browser will cause the application to reload in the first scenario, while the second will instruct the router to navigate at runtime. In our case, when the browser URL contains the `products` path, the router renders the `product-list` component on the page. On the contrary, when the application navigates to `products` by code, the router follows the same procedure and updates the browser URL.

If the user tries to navigate to a URL that does not match any route, Angular activates a custom type of route called **wildcard** or **fallback**. The wildcard route has a `path` property with two asterisks and matches any URL. The `component` property for this is usually an application-specific `PageNotFoundComponent` or the main component of the application.

Rendering components

The template of the main application component contains the `<router-outlet>` element, which is one of the main directives of the Angular router. It resides inside the `app.component.html` file and is used as a placeholder for components activated with routing. These components are rendered as a sibling element of the `<router-outlet>` element.

We have covered the basics and provided a minimal router setup. In the next section, we will look at a more realistic example and expand our knowledge of routing.

Configuring the main routes

When we start designing the architecture of an Angular application with routing, it is easiest to think about its main features, such as menu links that users can click to access. Products and shopping carts are basic features of the e-shop application we are currently building. Adding links and configuring them to activate certain features of an Angular application is part of the route configuration of the application.



You will need the source code of the Angular application we created in *Chapter 8, Communicating with Data Services over HTTP*, to follow along with the rest of the chapter. After you get the code, we suggest you take the following actions for simplicity:

- Remove the `auth.interceptor.ts` and its unit test file. Actual calls in the Fake Store API do not need authentication.
- Modify the `app.config.ts` file so that the `provideHttpClient` method does not use the interceptor.

To set up the route configuration of our application, we need to follow the steps below:

1. Run the following command to create a new Angular component for the shopping cart:

```
ng generate component cart
```

2. Open the `app.routes.ts` file and add the following import statements:

```
import { CartComponent } from './cart/cart.component';
import { ProductListComponent } from './product-list/product-list.component';
```

3. Add two route definition objects in the `routes` variable:

```
export const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: 'cart', component: CartComponent }
];
```

In the preceding snippet, the `products` route will activate the `ProductListComponent`, and the `cart` route will activate the `CartComponent`.

4. Open the `app.component.html` file and modify the `<header>` HTML element as follows:

```
<header>
  <h2>{{ settings.title }}</h2>
  <span class="spacer"></span>
  <div class="menu-links">
    <a routerLink="/products">Products</a>
    <a routerLink="/cart">My Cart</a>
  </div>
```

```
<app-auth></app-auth>  
</header>
```

In the preceding template, we apply the `routerLink` directive to anchor HTML elements and assign the route path we want to navigate. Notice that the path should start with `/` as opposed to the `path` property in the route definition object.

How the path starts depends on whether we want to use absolute or relative routing in our application, as we will learn later in the chapter.

5. Move the `<router-outlet>` HTML element inside the `<div>` element with the `content` class selector and remove the `<app-product-list>` component:

```
<main class="main">  
  <div class="content">  
    <router-outlet />  
  </div>  
</main>
```

6. Open the `app.component.ts` file, remove any references to the `ProductListComponent` class, and import the `RouterLink` class:

```
import { Component, inject } from '@angular/core';  
import { RouterLink, RouterOutlet } from '@angular/router';  
import { CopyrightDirective } from './copyright.directive';  
import { APP_SETTINGS } from './app.settings';  
import { AuthComponent } from './auth/auth.component';  
  
@Component({  
  selector: 'app-root',  
  imports: [  
    RouterOutlet,  
    RouterLink,  
    CopyrightDirective,  
    AuthComponent  
  ],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'  
})
```

7. Open the `app.component.css` file and replace every CSS style related to the `.social-links` selector with the following styles:

```
header {  
  display: flex;  
  flex-direction: row;  
  gap: 0.73rem;  
  justify-content: end;  
  margin-top: 1.5rem;  
}  
  
.menu-links {  
  display: flex;  
  align-items: center;  
  gap: 0.73rem;  
}  
  
.menu-links a {  
  transition: fill 0.3s ease;  
  color: var(--gray-400);  
}  
  
.menu-links a:hover {  
  color: var(--gray-900);  
}
```

8. Finally, open the `global-styles.css` file and add the following CSS styles:

```
a {  
  text-decoration: none;  
}  
.spacer {  
  flex: 1 1 auto;  
}
```

We are now ready to preview our Angular application:

1. Run the `ng serve` command and navigate to `http://localhost:4200`. Initially, the application page displays the application header and the copyright information only.

2. Click on the **Products** link. The application should display the product list and update the browser URL to match the /products path.
3. Now navigate to the root path at `http://localhost:4200` and append the /cart path at the end of the browser URL. The application should replace the product list view with the cart component:

cart works!



Routing in Angular works bi-directionally. It enables us to navigate to an Angular component using the in-app links or the browser address bar.

Congratulations! Your Angular application now supports in-app navigation.

We have barely scratched the surface of Angular routing. There are many features for us to investigate in the following sections. For now, let's try to break our components into more routes so that we can manage them easily.

Organizing application routes

Our application displays the product list along with the product details and the product create components. We need to organize the routing configuration so that different routes activate each component.

In this section, we will add a new route for the product create component. Later, in the *Passing parameters to routes* section, we will add a separate route for the product details component.

Let's get started with the product create component:

1. Open the `app.routes.ts` file and add the following import statement:

```
import { ProductCreateComponent } from './product-create/product-create.component';
```

2. Add the following route definition object in the `routes` variable:

```
{ path: 'products/new', component: ProductCreateComponent }
```

3. Open the `product-list.component.ts` file and remove any references to the `ProductCreateComponent` class.
4. Open the `product-list.component.html` file and remove the `<app-product-create>` element.

- Run the `ng serve` command to start the application, click on the **Products** link, and verify that the product create form is not displayed.

Currently, the product create component is only accessible using the browser URL, and we cannot reach it using the application UI. In the following section, we will learn how to accomplish that task and imperatively navigate to a route.

Navigating imperatively to a route

The product create component can only be activated by entering the address `http://localhost:4200/products/new` in the browser address bar. Let's add a button in the product list that will navigate us from the UI also:

- Open the `product-list.component.html` file and modify the second `@if` block as follows:



The `<path>` element below might be tricky to type out manually. Alternatively, you can find the code in the `ch09` folder in the book's GitHub repository and copy it from there.

```
@if (products) {  
  <div class="caption">  
    <h1>Products ({products.length})</h1>  
    <a routerLink="new">  
      <svg  
        width="24"  
        height="24"  
        xmlns="http://www.w3.org/2000/svg"  
        fill-rule="evenodd"  
        clip-rule="evenodd">  
        <path d="M11.5 0c6.347 0 11.5 5.153 11.5 11.5s-5.153 11.5  
        11.5 11.5-11.5-5.153-11.5-11.5 5.153-11.5 11.5-11.5zm0 1c5.795 0  
        10.5 4.705 10.5 10.5s-4.705 10.5-10.5 10.5-10.5-4.705-10.5-10.5  
        4.705-10.5 10.5-10.5zm.5 10h6v1h-6v6h-1v-6h-6v-1h6v-6h1v6z"/>  
      </svg>  
    </a>  
  </div>  
}
```

In the preceding snippet, we added an anchor element that will navigate us to the product create component, as indicated by the value of the `routerLink` directive.



The value of the `routerLink` directive is `new` and not `/products/new` as someone would expect. The preceding behavior is because the button resides in the product list component, which is already activated by the `products` part of the route.

The Angular router can synthesize the destination route by all activated routes, but if you don't want to start from the root, you can add a `/` before the route.

2. Open the `product-list.component.css` file and add the following CSS styles:

```
.caption {  
  display: flex;  
  align-items: center;  
  gap: 1.25rem;  
}  
  
path {  
  transition: fill 0.3s ease;  
  fill: var(--gray-400);  
}  
  
a:hover svg path {  
  fill: var(--gray-900);  
}
```

3. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { RouterLink } from '@angular/router';
```

4. Add the `RouterLink` class in the `imports` array of the `@Component` decorator:

```
@Component({  
  selector: 'app-product-list',  
  imports: [  
    ProductDetailComponent,  
    SortPipe,  
    AsyncPipe,  
    RouterLink
```

```
  ],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

5. Open the `product-create.component.css` file and add the following CSS style:

```
:host {
  width: 400px;
}
```

In the preceding style, the `:host` selector targets the host element of the product create component.

6. Run the `ng serve` command to start the application and navigate to `http://localhost:4200/products`:

Products (10)

 Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops

 John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet

 Mens Casual Premium Slim Fit T-Shirts

 Mens Casual Slim Fit

 Mens Cotton Jacket

 Pierced Owl Rose Gold Plated Stainless Steel Double

 SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s

 Solid Gold Petite Micropave

 WD 2TB Elements Portable External Hard Drive - USB 3.0

 White Gold Plated Princess

Figure 9.3: Product list

7. Click the button with the plus sign. The application redirects you to the /products/new route and activates the product create component:

Add new product

Title

Price

Category

Select a category

Create

Figure 9.4: Product create form

Although the product create component remains functional, our change introduced a flaw in the application's UX. The user does not have a visual indication when a new product is created because the product list belongs to a different route. We must modify the logic of the **Create** button so that it redirects the user to the product list upon successful creation of a product:

1. Open the `product-create.component.ts` file and add the following import statement:

```
import { Router } from '@angular/router';
```

2. Inject the Router service in the constructor of the `ProductCreateComponent` class:

```
constructor(private productsService: ProductsService, private router: Router) {}
```

3. Modify the `createProduct` method as follows:

```
createProduct(title: string, price: string, category: string) {
  this.productsService.addProduct({
    title,
    price: Number(price),
    category
  }).subscribe(() => this.router.navigate(['/products']));
}
```

In the preceding method, we call the `navigate` method of the `Router` service to navigate into the `/products` route of the application.



We use the `/` character because we are using absolute routing by default.

It accepts a **link parameters array** containing the destination route path we want to navigate.

4. Open the `products.service.ts` file and modify the `getProducts` method so that it uses the Fake Store API when there is no local product data:

```
getProducts(): Observable<Product[]> {
  if (this.products.length === 0) {
    const options = new HttpParams().set('limit', 10);
    return this.http.get<Product[]>(this.productsUrl, {
      params: options
    }).pipe(map(products => {
      this.products = products;
      return products;
    }));
  }
  return of(this.products);
}
```

If we do not make the preceding change, the product list component will always return data from the Fake Store API.

Our application now redirects users to the product list whenever they create a new product so that they can see it on the list.

So far, we have configured the application routing to activate components according to a given path. However, our application does not show any components in the following situations:

- When we navigate to the root path of the application
- When we try to navigate to a non-existing route

In the following section, we will learn how to use the built-in route paths that Angular router provides and improve the application UX.

Using built-in route paths

When we want to define a component that will be loaded when we navigate to the root path, we create a route definition object and set the `path` property to an empty string. A route with an empty string `path` is called the **default** route of the Angular application.

In our case, we want the default route to display the product list component. Open the `app.routes.ts` file and add the following route at the end of the `routes` variable:

```
{ path: '', redirectTo: 'products', pathMatch: 'full' }
```

In the preceding snippet, we tell the router to redirect to the `products` path when the application navigates to the default route. The `pathMatch` property tells the router how to match the URL to the `root` path property. In this case, the router redirects to the `products` path only when the URL matches the `root` path, which is the empty string.

If we run the application, we will notice that when the browser URL points to the root path of our application, we are redirected to the `products` path, and the product list is displayed on the screen.



We added the default route after all other routes because the order of the routes is important. The router selects routes with a first-match-wins strategy. More specific routes should be defined before less specific ones.

We have encountered the concept of unknown routes in the *Introducing the Angular router* section. We saw briefly how to set up a **wildcard** route to display a `PageNotFoundComponent` when our application tries to navigate to a route that does not exist. In real-world applications, it is common to create such a component, especially if you want to display additional information to the user, such as what next steps they can follow. In our case, which is simpler, we will redirect to the `products` route.

Open the `app.routes.ts` file and add the following route at the end of the `routes` variable:

```
{ path: '**', redirectTo: 'products' }
```



The wildcard route must be the last entry in the route list because the application should only reach it if there are no matching routes.

If we run our application using the `ng serve` command and navigate to an unknown path, our application will display the product list.

Until now, we have relied on the address bar of the browser to indicate which route is active at any given time. As we will learn in the following section, we could improve the user experience using CSS styling.

Styling router links

The application header contains the **Products** and the **My Cart** links. When we navigate to each one, it is not clear which route has been activated. The Angular router exports the `routerLinkActive` directive, which we can use to change the style of a link when the corresponding route is active. It works similarly to the class binding we learned about in *Chapter 3, Structuring User Interfaces with Components*. It accepts a list of class names or a class that is added when the link is active and removed when it becomes inactive.

Let's see how to use it in our application:

1. Open the `app.component.css` file and add the following CSS style:

```
.menu-links a.active {  
  color: var(--electric-violet);  
}
```

2. Open the `app.component.ts` file and import the `RouterLinkActive` class from the `@angular/router` npm package:

```
import { RouterLink, RouterLinkActive, RouterOutlet } from '@  
angular/router';
```

3. Add the `RouterLinkActive` class in the `imports` array of the `@Component` decorator:

```
@Component({  
  selector: 'app-root',  
  imports: [  
    RouterOutlet,  
    RouterLink,  
    RouterLinkActive,  
    CopyrightDirective,  
    AuthComponent  
,  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css'  
])
```

4. Open the `app.component.html` file and add the `routerLinkActive` directive to both links:

```
<div class="menu-links">
  <a routerLink="/products" routerLinkActive="active">Products</a>
  <a routerLink="/cart" routerLinkActive="active">My Cart</a>
</div>
```

Now, when we click on an application link in the header, its color changes to denote the link is active.

We have learned how to use routing and activate components that do not need any parameters. However, the product details component accepts the product ID as a parameter. In the next section, we will learn how to activate the component using dynamic route parameters.

Passing parameters to routes

A common scenario in enterprise web applications is to have a list of items, and when you click on one of them, the page changes the current view and displays details of the selected item. The previous approach resembles a master-detail browsing functionality, where each generated URL on the master page contains the identifiers required to load each item on the detail page.

We can represent the previous scenario with two routes navigating to different components. One component is the list of items, and the other is the item details. So, we need to find a way to create and pass dynamic item-specific data from one route to the other.

We are tackling double trouble here: creating URLs with dynamic parameters at runtime and parsing the value of these parameters. No problem: the Angular router has our back, and we will see how with a real example.

Building a detail page using route parameters

The product list in our application currently displays a list of products. When we click on a product, the product details appear below the list. We need to refactor the previous workflow so that the component responsible for displaying product details is rendered on a different page from the list. We will use the Angular router to redirect the user to the new page upon clicking on a product from the list.

The product list component currently passes the selected product ID via input binding. We will use the Angular router to pass the product ID as a route parameter instead:

1. Open the `app.routes.ts` file and add the following `import` statement:

```
import { ProductDetailComponent } from './product-detail/product-detail.component';
```

2. Add the following route definition in the `routes` variable after the `products/new` route:

```
{ path: 'products/:id', component: ProductDetailComponent }
```

The colon character denotes `id` as a route parameter in the new route definition object. If a route has multiple parameters, we separate them with `/`. As we will learn later, the parameter name is important when we want to consume its value in our components.

3. Open the `product-list.component.html` file and add an anchor element for the product title so that it uses the new route definition:

```
<ul class="pill-group">
  @for (product of products | sort; track product.id) {
    <li class="pill" (click)="selectedProduct = product">
      @switch (product.category) {
        @case ('electronics') { 🖥 }
        @case ('jewelery') { 💎 }
        @default { 🔍 }
      }
      <a [routerLink]=[product.id]>{{product.title}}</a>
    </li>
  } @empty {
    <p>No products found!</p>
  }
</ul>
```

In the preceding snippet, the `routerLink` directive uses property binding to set its value in a link parameters array. We pass the `id` of the product template reference variable as a parameter in the array.



We do not need to prefix the value of the link parameters array with `/products` because that route already activates the product list.

4. Remove the `<app-product-detail>` component and the `click` event binding from the `` tag.



We can refactor the `product-list.component.ts` file and remove any code that uses the `selectedProduct` property and the `ProductDetailComponent` class. The product list does not need to keep the selected product in its local state because we are navigating away from the list upon choosing a product.

We can now proceed by modifying the product detail component so that it works with routing:

1. Open the `product-detail.component.css` file and add a CSS style to set the width of the host element:

```
:host {  
  width: 450px;  
}
```

2. Open the `product-detail.component.ts` file and modify the import statements as follows:

```
import { CommonModule } from '@angular/common';  
import { Component, input, OnInit } from '@angular/core';  
import { ActivatedRoute, Router } from '@angular/router';  
import { Product } from '../product';  
import { Observable, switchMap } from 'rxjs';  
import { ProductsService } from '../products.service';  
import { AuthService } from '../auth.service';
```

The Angular router exports the `ActivatedRoute` service, which we can use to retrieve information about the currently activated route, including any parameters.

3. Modify the component constructor to inject the `ActivatedRoute` and `Router` services:

```
constructor(  
  private productService: ProductsService,  
  public authService: AuthService,  
  private route: ActivatedRoute,  
  private router: Router  
) {}
```

4. Modify the list of implemented interfaces of the ProductDetailComponent class:

```
export class ProductDetailComponent implements OnInit
```

5. Create the following ngOnInit method:

```
ngOnInit(): void {
  this.product$ = this.route.paramMap.pipe(
    switchMap(params => {
      return this.productService.getProduct(Number(params.get('id')));
    })
  );
}
```

The ActivatedRoute service contains the paramMap observable, which we can use to subscribe and get route parameter values. The switchMap RxJS operator is used when we want to get a value from an observable, complete it, and pass the value down to another observable. We use it, in this case, to pipe the id parameter from the paramMap observable to the getProduct method of the ProductsService class.

6. Modify the changePrice and remove methods so that the application will redirect to the product list upon completion of each action:

```
changePrice(product: Product, price: string) {
  this.productService.updateProduct(product.id, Number(price)).
  subscribe(() => {
    this.router.navigate(['/products']);
  });
}

remove(product: Product) {
  this.productService.deleteProduct(product.id).subscribe(() => {
    this.router.navigate(['/products']);
  });
}
```

7. Remove the ngOnChanges method because the component and its bindings are initialized every time the route is activated.

8. Remove the output event emitters because the product list component is not a parent component anymore. Leave the `id` input property as is because we will use it later in the chapter.
9. Leave the `addToCart` method empty for now. We will use it later in *Chapter 10, Collecting User Data with Forms*.

It is also worth noting the following:

1. The `paramMap` observable returns an object of the `ParamMap` type. We can use the `get` method of the `ParamMap` object to pass the parameter name we defined in the route configuration and access its value.
2. We convert the value of the `id` parameter to a number because route parameter values are always strings.

If we run the application using the `ng serve` command and click on a product from the list, the application navigates us to the product details component:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III
6 Gb/s**

€109.00

New price

Change

electronics

Delete

Figure 9.5: Product Details page



If you refresh the browser, the application will not display the product because the `getProduct` method of the `ProductsService` class works only with the cached version of product data. You must go to the product list again and select a product because the local cache has been reset. Note that this behavior is based on the current implementation of the e-shop application and is not tied to the Angular router architecture.

In the previous example, we used the `paramMap` property to get route parameters as an observable. So, ideally, our component could be notified of new values during its lifetime. But the component is destroyed each time we want to select a different product from the list, and so is the subscription to the `paramMap` observable.

Alternatively, we can avoid using observables by reusing the instance of a component as soon as it remains rendered on the screen during consecutive navigations. We can achieve this behavior using child routes, as we will learn in the following section.

Reusing components using child routes

Child routes are a perfect solution when we want a landing page component that will provide routing to other components. The component should contain a `<router-outlet>` element in which child routes will be loaded.

Suppose that we want to define the layout of our Angular application like this:

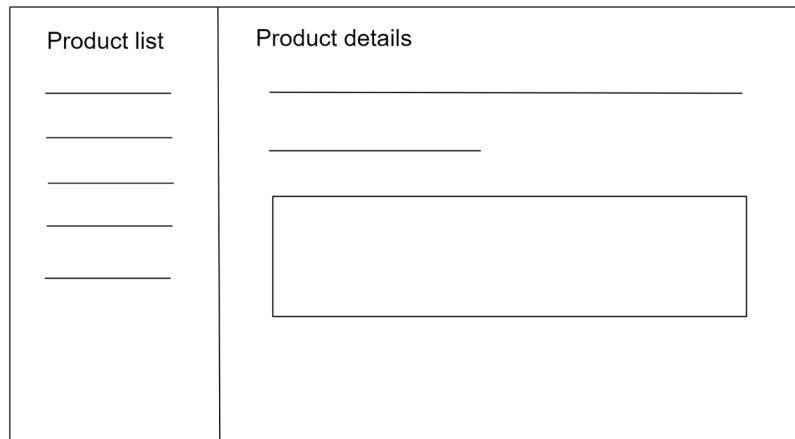


Figure 9.6: Master-detail layout

The scenario in the previous diagram requires the product list component to contain a `<router-outlet>` element to render the product details component when the related route is activated.

The product details component will be rendered in the `<router-outlet>` of the product list component and not in the `<router-outlet>` of the main application component.

The product details component is not destroyed when we navigate from one product to another. Instead, it remains in the DOM tree, and its `ngOnInit` method is called once, the first time we select a product. When we choose a new product from the list, the `paramMap` observable emits the `id` of the new product. The new product is fetched using the `ProductsService` class, and the component template is refreshed to reflect the new changes.

The route configuration of the application, in this case, would be as follows:

```
export const routes: Routes = [
  {
    path: 'products',
    component: ProductListComponent,
    children: [
      { path: 'new', component: ProductCreateComponent },
      { path: ':id', component: ProductDetailComponent },
    ]
  },
  { path: 'cart', component: CartComponent },
  { path: '', redirectTo: 'products', pathMatch: 'full' },
  { path: '**', redirectTo: 'products' }
];
```

In the preceding snippet, we use the `children` property of the route definition object to define child routes containing a list of route definition objects.



Notice also that we removed the word `products` from the `path` property of the children routes because the parent route will append it.

A parent route can also provide services to its children by using the `providers` property of the route definition object. Providing services in a route is very helpful when we want to limit access to a subset of the routing configuration. If we wanted to restrict the `ProductsService` class only to the product-related components, we should do the following:

```
{
  path: 'products',
  component: ProductListComponent,
  children: [
    { path: 'new', component: ProductCreateComponent },
```

```
{ path: ':id', component: ProductDetailComponent },
],
providers: [ProductsService]
}
```

Angular creates a separate injector when providing services in route definition objects, which is an immediate child of the root injector. Suppose the service is also provided in the root injector, and suppose the cart component uses that. In that case, the instance created by one of the product-related components will differ from that of the cart component.

We have learned how to use the `paramMap` observable in Angular routing. In the following section, we will discuss an alternative approach using snapshots.

Taking a snapshot of route parameters

When we select a product from the list, the product list component is removed from the DOM tree, and the product details component is added. To choose a different product, we need to click on either the **Products** link or the back button of our browser. Consequently, the product details component is replaced by the product list component in the DOM. So, we are in a situation where only one component is displayed on the screen at any time.

When the product details component is destroyed, so is its `ngOnInit` method and the subscription to the `paramMap` observable. So, we do not benefit from using observables at this point. Alternatively, we could use the `snapshot` property of the `ActivatedRoute` service to get values for route parameters, as follows:

```
ngOnInit(): void {
  const id = this.route.snapshot.params['id'];
  this.product$ = this.productService.getProduct(id);
}
```

The `snapshot` property represents the current value of a route parameter, which also happens to be the initial value. It contains the `params` property, an object of route parameter key-value pairs we can access.



If you are sure your component will not be reused, use the snapshot approach.

So far, we have dealt with routing parameters in the form of `products/:id`. We use these parameters to navigate to a component that requires the parameter. In our case, the product details component requires the `id` parameter to get specific product details. However, there is another type of route parameter when we need it to be optional, as we will learn in the following section.

Filtering data using query parameters

In *Chapter 8, Communicating with Data Services over HTTP*, we learned how to pass query parameters to a request using the `HttpParams` class. The Angular router also supports passing query parameters through the application's URL.

The `getProducts` method in the `products.service.ts` file uses HTTP query parameters to limit product results returned from the Fake Store API:

```
getProducts(): Observable<Product[]> {
  if (this.products.length === 0) {
    const options = new HttpParams().set('limit', 10);
    return this.http.get<Product[]>(this.productsUrl, {
      params: options
    }).pipe(map(products => {
      this.products = products;
      return products;
    }));
  }
  return of(this.products);
}
```

It uses a hardcoded value for setting the `limit` query parameter. We will modify the application so that the product list component passes the `limit` value dynamically:

1. Open the `products.service.ts` file and modify the `getProducts` method so that the `limit` is passed as a parameter:

```
getProducts(limit?: number): Observable<Product[]> {
  if (this.products.length === 0) {
    const options = new HttpParams().set('limit', limit || 10);
    return this.http.get<Product[]>(this.productsUrl, {
      params: options
    }).pipe(map(products => {
```

```
        this.products = products;
        return products;
    }));
}
return of(this.products);
}
```

In the preceding method, if the `limit` value is `falsy`, we pass a default value of `10` to the `query` parameter.



A `falsy` value evaluates to `False` in a Boolean context and can be `null`, `undefined`, `0`, or `False`. You can read more at <https://developer.mozilla.org/docs/Glossary/Falsy>.

2. Open the `product-list.component.ts` file and import the `ActivatedRoute` service and the `switchMap` RxJS operator:

```
import { RouterLink, ActivatedRoute } from '@angular/router';
import { Observable, switchMap } from 'rxjs';
```

3. Inject the `ActivatedRoute` service in the constructor of the `ProductListComponent` class:

```
constructor(private productService: ProductsService, private route: ActivatedRoute) {}
```

4. The `ActivatedRoute` service contains a `queryParamMap` observable that we can subscribe to get query parameter values. It returns a `ParamMap` object, similar to the `paramMap` observable we saw earlier, which we can query to get parameter values. Modify the `getProducts` method to use the `queryParamMap` observable:

```
private getProducts() {
    this.products$ = this.route.queryParamMap.pipe(
        switchMap(params => {
            return this.productService.getProducts(Number(params.get('limit')));
        })
    );
}
```

In the preceding snippet, we use the `switchMap` RxJS operator to pipe the `limit` parameter from the `queryParamMap` observable to the `getProducts` method of the `ProductsService` class as a number.

5. Run the `ng serve` command to start the application and navigate to `http://localhost:4200?limit=5`. You should see a list of 5 products:

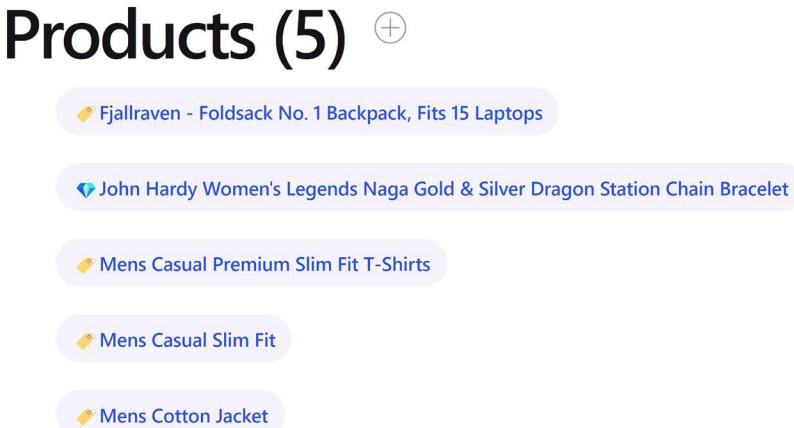


Figure 9.7: Filtered product list

Try to experiment with different values for the `limit` parameter and observe the output.

Query parameters in routing are powerful and can be used for various use cases, such as filtering and sorting data. They can also be used when working with snapshot-based routing.

In the following section, we will explore a new innovative way to pass route parameters using component input properties.

Binding input properties to routes

We have already learned, in *Chapter 3, Structuring User Interfaces with Components*, that we use input and output bindings to inter-communicate between components. An input binding can also pass route parameters while navigating to a component. We will see an example using the product detail component:

1. The input binding with route parameters is not enabled by default in the Angular router. We must activate it from the application configuration file. Open the `app.config.ts` file and import the `withComponentInputBinding` function from the `@angular/router` npm package:

```
import { provideRouter, withRouterComponentInputBinding } from '@angular/router';
```

2. Pass the preceding function as the second parameter in the provideRouter method:

```
provideRouter(routes, withRouterComponentInputBinding()),
```

3. Now, open the product-detail.component.ts file and change the type of the id component property to a string:

```
id = input<string>();
```

We must change the property type because routing parameters are passed as strings.

4. Modify the ngOnInit method to use the id parameter to fetch a product:

```
ngOnInit(): void {
  this.product$ = this.productService.getProduct(Number(this.id()));
}
```

5. Run the ng serve command and verify that the product details are displayed upon selecting a product from the list.

Binding route parameters to component input properties has the following advantages:

- The TypeScript component class is simpler because we do not have asynchronous calls with observables
- We can access existing components that work with input and output bindings using a route



Input binding works with components that are activated via routing. If we want to access any route parameter from another component, we must use the ActivatedRoute service.

Now that we have learned all the different ways to pass parameters during navigation, we have covered all the essential information we need to start building Angular applications with routing. In the following sections, we will focus on advanced practices that enhance the user experience when using in-app navigation in Angular applications.

Enhancing navigation with advanced features

So far, we have covered basic routing with route and query parameters. The Angular router is quite capable, though, and able to do much more, such as the following:

- Controlling access to a route
- Preventing navigation away from a route
- Prefetching data to improve application UX
- Lazy-loading routes to speed up response time

In the following sections, we will learn about all these techniques in more detail.

Controlling route access

When we want to control access to a particular route, we use a **guard**. To create a guard, we use the `ng generate` command of the Angular CLI, passing the word `guard` and its name as parameters:

```
ng generate guard auth
```

When we execute the previous command, the Angular CLI asks what type of guard we would like to create. There are multiple types of guards that we can create according to the functionality that they provide:

- `CanActivate`: Controls whether a route can be activated
- `CanActivateChild`: Controls whether children routes can be activated
- `CanDeactivate`: Controls whether a route can be deactivated



Deactivation happens when we navigate away from a route.

- `CanMatch`: Controls whether a route can be accessed at all

Select `CanActivate` and press *Enter*. The Angular CLI creates the following `auth.guard.ts` file:

```
import { CanActivateFn } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) => {
  return true;
};
```

The guard that we created is a function of type `CanActivateFn`, which accepts two parameters:

- `route`: Indicates the route that will be activated
- `state`: Contains the state of the router upon successful navigation



The `CanActivateFn` function can return a boolean value, either synchronously or asynchronously. In the latter case, the router will wait for the observable or the promise to resolve before continuing. If the asynchronous event does not complete, the navigation will not continue. It can also return a `UrlTree` object, which will cause new navigation to a defined route.

Our guard returns `true` immediately, allowing free access to the route. Let's add custom logic to control access based on whether the user is logged in:

1. Modify the `import` statements as follows:

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';
```

2. Replace the body of the arrow function with the following snippet:

```
const authService = inject(AuthService);
const router = inject(Router);

if (authService.isLoggedIn()) {
  return true;
}
return router.parseUrl('/');
```

In the preceding snippet, we use the `inject` method to inject the `AuthService` and `Router` services into the function. We then check the value of the `isLoggedIn` signal. If it is `true`, we allow the application to navigate to the requested route. Otherwise, we use the `parseUrl` method of the `Router` service to navigate to the root path of the Angular application.



The `parseUrl` method returns a `UrlTree` object, which effectively cancels the previous navigation and redirects the user to the URL passed in the parameter. It is advised to use it over the `navigate` method, which may introduce unexpected behavior and can lead to complex navigation issues.

3. Open the `app.routes.ts` file and add the following `import` statement:

```
import { authGuard } from './auth.guard';
```

4. Add the `authGuard` function in the `canActivate` array of the `cart` route:

```
{
  path: 'cart',
  component: CartComponent,
  canActivate: [authGuard]
}
```



The `canActivate` property is an array because multiple guards can control route activation. The order of guards in the array is important. If one of the guards fails to pass, Angular will prevent access to the route.

Only authenticated users can now access the shopping cart. If you run the application using the `ng serve` command and click the `My Cart` link, you will notice that nothing happens.



When you try to access the shopping cart from the product list, you always remain on the same page. This is because the redirection that happens due to the authentication guard does not have any effect when you are already in the redirected route.

Another guard type related to the activation of a route is the `CanDeactivate` guard. In the following section, we will learn how to use it to prevent a user leaving a route.

Preventing navigation away from a route

A guard that controls if a route can be deactivated is a function of the `CanDeactivateFn` type. We will learn how to use it by implementing a guard that notifies the user of pending products in the cart when they navigate away from the cart component:

1. Run the following command to generate a new guard:

```
ng generate guard checkout
```

2. Select the `CanDeactivate` type from the list and press `Enter`.
3. Open the `checkout.guard.ts` file and add the following `import` statement:

```
import { CartComponent } from './cart/cart.component';
```

4. Change the generic of the `CanDeactivateFn` to `CartComponent` and remove the parameters of the arrow function.



In a real-world scenario, we will probably need to add more components in the generics to create a generic guard.

5. Replace the body of the arrow function with the following snippet:

```
const confirmation = confirm(  
  'You have pending items in your cart. Do you want to continue?'  
);  
return confirmation;
```

In the preceding snippet, we use the `confirm` method of the global `window` object to display a confirmation dialog before navigating away from the cart component. The application execution will wait until the confirmation dialog is dismissed as a user interaction.

6. Open the `app.routes.ts` file and add the following `import` statement:

```
import { checkoutGuard } from './checkout.guard';
```

7. A route definition object contains a `canDeactivate` array similar to `canActivate`. Add the `checkoutGuard` function to the `canDeactivate` array of the `cart` route:

```
{  
  path: 'cart',  
  component: CartComponent,  
  canActivate: [authGuard],  
  canDeactivate: [checkoutGuard]  
}
```



The `canDeactivate` property is an array because multiple guards can control route deactivation. The order of guards in the array is important. If one of the guards fails to pass, Angular will prevent a user leaving the route.

For such a simple scenario, we could have written the logic of the `checkoutGuard` function inline to avoid the creation of the `checkout.guard.ts` file:

```
{
```

```

path: 'cart',
component: CartComponent,
canActivate: [authGuard],
canDeactivate: [() => confirm('You have pending items in your cart. Do you want to continue?')]
}

```

Run the application using the `ng serve` command and click the **My Cart** link after you have logged in. If you then click on the **Products** link or press the back button of the browser, you should see a dialog with the following message:

You have pending items in your cart. Do you want to continue?

If you click the **Cancel** button, the navigation is canceled, and the application remains in its current state. If you click the **OK** button, you will be redirected to the product list.

Prefetching route data

You may have noticed that when you navigate to the root path of the application for the first time, there is a delay in displaying the product list. It is reasonable since we are making an HTTP request to the backend API. However, the product list component was already initialized at that time.

The preceding behavior may lead to unwanted effects if the component contains logic that interacts with data during initialization. To solve this problem, we can use a **resolver** to prefetch the product list and load the component when data are available.



A resolver can be handy when handling possible errors before activating a route. It would be more appropriate to navigate to an error page if the request to the API does not succeed instead of displaying a blank page.

To create a resolver, we use the `ng generate` command of the Angular CLI, passing the word `resolver` and its name as parameters:

```
ng generate resolver products
```

The preceding command creates the following `products.resolver.ts` file:

```

import { ResolveFn } from '@angular/router';

export const productsResolver: ResolveFn<boolean> = (route, state) => {
  return true;
};

```

The resolver that we created is a function of type `ResolveFn`, which accepts two parameters:

- `route`: Indicates the route that will be activated
- `state`: Contains the state of the activated route



A `ResolveFn` function can return an observable or promise. The router will wait for the observable or the promise to resolve before continuing. If the asynchronous event does not complete, the navigation will not continue.

Currently, our resolver returns a boolean value. Let's add custom logic so that it returns an array of products:

1. Add the following `import` statements:

```
import { inject } from '@angular/core';
import { Product } from './product';
import { ProductsService } from './products.service';
```

2. Modify the `productsResolver` function so that it returns a product array:

```
export const productsResolver: ResolveFn<Product[]> = (route, state)
=> {
  return [];
};
```

3. Use the `inject` method to inject `ProductsService` in the function body:

```
const productService = inject(ProductsService);
```

4. Use the `queryParamMap` property to get the `limit` parameter value from the current route:

```
const limit = Number(route.queryParamMap.get('limit'));
```

5. Replace the `return` statement with the following:

```
return productService.getProducts(limit);
```

6. The resulting function should look like the following:

```
export const productsResolver: ResolveFn<Product[]> = (route, state)
=> {
  const productService = inject(ProductsService);
  const limit = Number(route.queryParamMap.get('limit'));
```

```
    return productService.getProducts(limit);
};
```

Now that we have created the resolver, we can connect it with the product list component:

1. Open the `app.routes.ts` file and add the following `import` statement:

```
import { productsResolver } from './products.resolver';
```

2. Add the following `resolve` property to the `products` route:

```
{
  path: 'products',
  component: ProductListComponent,
  resolve: {
    products: productsResolver
  }
}
```

The `resolve` property is an object that contains a unique name as a key and the resolver function as a value. The key name is important because we will use it in our components to access the resolved data.

3. Open the `product-list.component.ts` file and import the `of` operator from the `rxjs` npm package:

```
import { Observable, switchMap, of } from 'rxjs';
```

4. Modify the `getProducts` method so that it subscribes to the `data` property of the `ActivatedRoute` service:

```
private getProducts() {
  this.products$ = this.route.data.pipe(
    switchMap(data => of(data['products']))
  );
}
```

In the preceding snippet, the `data` observable emits an object whose value exists in the `products` key. Notice that we use the `switchMap` operator to return `products` in a new observable.



At this point, we can also remove any references to the `ProductsService` class because it is not needed anymore.

5. Run the `ng serve` command to start the application and verify that the product list is displayed when navigating to `http://localhost:4200`.

Angular resolvers improve application performance when complex initialization logic exists in routed components. Another way to improve the application performance is to load components or child routes on demand, as we will learn in the following section.

Lazy-loading parts of the application

Our application may grow at some point, and the amount of data we put into it may also increase. The application may take a long time to start initially, or certain parts can take a long time to load. To overcome these problems, we can use a technique called **lazy loading**.

Lazy loading means we don't initially load certain application parts, such as Angular components or routes. There are many advantages of lazy loading in an Angular application:

- Components and routes can be loaded upon request from the user
- Users who visit certain areas of your application can significantly benefit from this technique
- We can add more features in a lazy-loaded area without affecting the overall application bundle size

To understand how lazy loading in Angular works, we will create a new component that displays the current user profile.



A good practice is to lazy-load parts of the application that are not used frequently, such as the profile of the currently logged-in user.

Let's get started:

1. Run the following command to create an Angular component:

```
ng generate component user
```

2. Create a file named `user.routes.ts` in the `src\app` folder and add the following content:

```
import { UserComponent } from './user/user.component';

export default [
  { path: '', component: UserComponent }
];
```

In the preceding snippet, we set the `path` property to an empty string to activate the route by default. We also use the `default` keyword to benefit from the default export feature in lazy loading.

3. Open the `app.routes.ts` file and add the following route definition in the `routes` variable:

```
{ path: 'user', loadChildren: () => import('./user.routes') }
```

The `loadChildren` property of a route definition object is used to lazy-load Angular routes. It returns an arrow function that uses a dynamic `import` statement to lazy-load the routes file. The `import` function accepts the relative path of the routes file we want to import.

4. Add a new anchor element to the `<header>` element of the `app.component.html` file that links to the newly created route:

```
<div class="menu-links">
  <a routerLink="/products" routerLinkActive="active">Products</a>
  <a routerLink="/cart" routerLinkActive="active">My Cart</a>
  <a routerLink="/user" routerLinkActive="active">My Profile</a>
</div>
```

5. Run the command `ng serve` and observe the output in the console window. It should look similar to the following:

Initial chunk files	Names	Raw size
polyfills.js	polyfills	82.71 kB
main.js	main	47.22 kB
styles.css	styles	1.14 kB
		Initial total 131.07 kB
Lazy chunk files	Names	Raw size
chunk-D3RURZVW.js	user-routes	1.26 kB

```
Application bundle generation complete. [1.234 seconds]
```

In the preceding output, we can see that the Angular CLI has created a lazy chunk file named **user-routes** in addition to the initial chunk files of the application.

6. Navigate with your browser to `http://localhost:4200` and open the developer tools.
7. Click the **My Profile** link and inspect the **Network** requests tab:

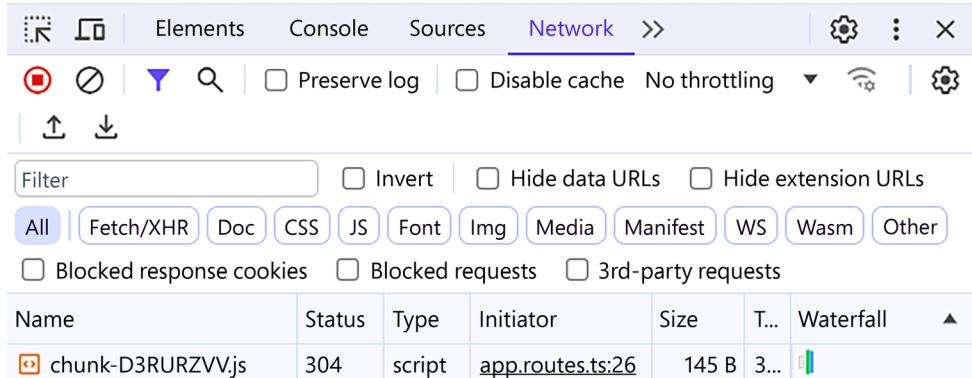


Figure 9.8: Lazy-loaded route

The application initiates a new request to the chunk file, which is the bundle of the user route. The Angular framework creates a new bundle for each lazy-loaded artifact and does not include it in the main application bundle.

If you navigate away and click the **My Profile** link again, you will notice that the application does not make a new request to load the bundle file. As soon as a lazy-loaded route is requested, it is kept in memory and can be used for subsequent requests.

Lazy loading works not only with routes but also with components. We could have lazy-loaded the user component instead of the whole route by modifying the user route as follows:

```
{  
  path: 'user',  
  loadComponent: () => import('./user/user.component').then(c =>  
    c.UserComponent),  
}
```

In the preceding snippet, we use the `loadComponent` property to import the `user.component.ts` file dynamically. The `import` function returns a promise that we chain with the `then` method to load the `UserComponent` class.

The user route is currently accessible for all users, even if not authenticated. In the following section, we will learn how to protect them using guards.

Protecting a lazy-loaded route

We can control unauthorized access to a lazy-loaded route similarly to how we can on normal routes. However, our guards need to support a function type named `CanMatchFn`.

We will extend our authentication guard for use with lazy-loaded routes:

1. Open the `auth.guard.ts` file and import the `CanMatchFn` type from the `@angular/router` npm package:

```
import { CanActivateFn, CanMatchFn, Router } from '@angular/router';
```

2. Modify the signature of the `authGuard` function as follows:

```
export const authGuard: CanActivateFn | CanMatchFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn()) {
    return true;
  }
  return router.parseUrl('/');
};
```

3. Open the `app.routes.ts` file and add the `authGuard` function in the `canMatch` array of the user route:

```
{
  path: 'user',
  loadChildren: () => import('./user.routes'),
  canMatch: [authGuard]
}
```



The `canMatch` property is an array because multiple guards can control route matching. The order of guards in the array is important. If one of the guards fails to match with a route, Angular will prevent access to the route.

If we now run the application and click the **My Profile** link, we will notice that we cannot navigate to the respective component unless we are authenticated.

Lazy loading is a technique preferred when the application performance is critical. Angular has also introduced a more performant feature to delay loading parts of an Angular application called **deferrable views**. Deferrable views give developers more fine-grained control over the conditions under which a part of the application will be loaded. We will explore deferrable views in *Chapter 15, Optimizing Application Performance*.

Summary

We have now uncovered the power of the Angular router, and we hope you have enjoyed this journey into the intricacies of this library. One of the things that shines in the Angular router is the vast number of options and scenarios we can cover with such a simple but powerful implementation.

We have learned the basics of setting up routing and handling different types of parameters. We have also learned about more advanced features, such as child routing. Furthermore, we have learned how to protect our routes from unauthorized access. Finally, we have shown the full power of routing and how you can improve response time with lazy loading and prefetching.

In the next chapter, we will beef up our application components to showcase the mechanisms underlying web forms in Angular and the best strategies to grab user input with form controls.

10

Collecting User Data with Forms

Web applications use forms to collect input data from users. Use cases vary, from allowing users to log in, fill in payment information, book a flight, or even perform a search. Form data can later be persisted on local storage or be sent to a server using a backend API.

In this chapter, we will cover the following topics about forms:

- Introducing web forms
- Building template-driven forms
- Building reactive forms
- Using a form builder
- Validating input in forms
- Manipulating form state

Technical requirements

The chapter contains various code samples to walk you through creating and managing forms in Angular. You can find the related source code in the ch10 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Introducing web forms

A form usually has the following characteristics that enhance the user experience of a web application:

- Defines different kinds of input fields

- Sets up different kinds of validations and displays validation errors to the user
- Supports different strategies for handling data if the form is in an error state

The Angular framework provides two approaches to handling forms: **template-driven** and **reactive**. Neither approach is considered better; you must choose the one that best suits your scenario. The main difference between the two approaches is how they manage data:

- **Template-driven forms:** These are easy to set up and add to an Angular application. They operate solely on the component template to create elements and configure validation rules; thus, they are not easy to test. They also depend on the change detection mechanism of the framework.
- **Reactive forms:** These are more robust when scaling and testing. They operate in the component class to manage input controls and set validation rules. They also manipulate data using an intermediate form model, maintaining their immutable nature. This technique is for you if you use reactive programming techniques extensively or if your Angular application comprises many forms.

A form in a web application consists of a `<form>` HTML element that contains HTML elements for entering data, such as `<input>` and `<select>` elements, and `<button>` elements for interacting with the data. The form can retrieve and save data locally or send it to a server for further manipulation. The following is an example of a simple form that is used for logging a user into a web application:

```
<form>
  <div>
    <input type="text" name="username" placeholder="Username" />
  </div>
  <div>
    <input type="password" name="password" placeholder="Password" />
  </div>
  <button type="submit">Login</button>
</form>
```

The preceding form has two `<input>` elements: one for entering the username and another for entering the password. The type of the `password` field is set to `password` so that the content of the input control is not visible while typing. The type of the `<button>` element is set to `submit` so that the form can collect data by a user clicking on the button or pressing `Enter` on any input control.



We could add another button with the `reset` type if we wanted to reset form data.

Notice that an HTML element must reside inside the `<form>` element to be part of it. The following screenshot shows what the form looks like when rendered on a page:

The form consists of three elements: a text input field with the placeholder "Username", a text input field with the placeholder "Password", and a single button labeled "Login".

Figure 10.1: Login form

Web applications can significantly enhance the user experience by using forms that provide features such as autocomplete in input controls or prompting the user to save sensitive data. Now that we have understood what a web form looks like, let's learn how all that fits into the Angular framework.

Building template-driven forms

Template-driven forms are one of two different ways of integrating forms with Angular. These can be powerful in cases where we want to create small and simple forms for our Angular application.

We learned about data binding in *Chapter 3, Structuring User Interfaces with Components*, and how we can use different types to read data from an Angular component and write data to it. In that case, binding is either one way or another, called **one-way binding**. In template-driven forms, we can combine both ways and create a **two-way binding** that can read and write data simultaneously. Template-driven forms provide the `ngModel` directive, which we can use in our components to get this behavior. To learn more about template-driven forms, we will convert the change price functionality of our product detail component to work with Angular forms.



You will need the source code of the Angular application we created in *Chapter 9, Navigating through Applications with Routing*, to follow along with the rest of the chapter.

Let's get started:

1. Open the `product-detail.component.ts` file and add the following import statement:

```
import { FormsModule } from '@angular/forms';
```

We add template-driven forms to an Angular application using the `FormsModule` class from the `@angular/forms` npm package.

2. Add `FormsModule` in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-detail',
  imports: [CommonModule, FormsModule],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css']
})
```

3. Open the `product-detail.component.html` file and modify the `<input>` element as follows:

```
<input placeholder="New price" type="number" name="price"
[(ngModel)]="product.price" />
```

In the preceding snippet, we bind the `price` property of the `product` template variable to the `ngModel` directive of the `<input>` element. The `name` attribute is required so that Angular can internally create a unique form control to distinguish it.



The syntax of the `ngModel` directive is known as a *banana in a box*, and we create it in two steps. First, we make the *banana* by surrounding `ngModel` in parentheses `()`. Then, we put it *in a box* by surrounding it with square brackets `[]`.

4. Modify the `<button>` element as follows:

```
<button class="secondary" type="submit">Change</button>
```

In the preceding snippet, we remove the `click` event from the `<button>` element because submitting the form will update the `price`. We also add the `submit` type to indicate that the form submission can happen by a user clicking the button.

5. Surround the `<input>` and `<button>` elements with the following `<form>` element:

```
<form (ngSubmit)="changePrice(product)">
  <input placeholder="New price" type="number" name="price"
  [(ngModel)]="product.price" />
  <button class="secondary" type="submit">Change</button>
</form>
```

In the preceding snippet, we bind the `changePrice` method to the `ngSubmit` event of the form. The binding will trigger the method execution if we press *Enter* inside the input box or click the button. The `ngSubmit` event is part of the `Angular FormsModule` and hooks on the native `submit` event of an HTML form.

6. Open the `product-detail.component.ts` file and modify the `changePrice` method as follows:

```
changePrice(product: Product) {
  this.productService.updateProduct(
    product.id,
    product.price
  ).subscribe(() => this.router.navigate(['/products']));
}
```

7. Run the application using the `ng serve` command and select a product from the list.
8. You will notice that the current product price is already displayed inside the input box. Try to change the price, and you will notice that the current price of the product is also changing while you type:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III
6 Gb/s**

€79.00

79

Change

electronics

Delete

Figure 10.2: Two-way binding

The behavior of our application depicted in the preceding image is the magic behind two-way binding and `ngModel`.



Two-way binding was the biggest selling point when AngularJS came out in 2010. It was complex to achieve that behavior in those days with vanilla JavaScript and jQuery.

While we type inside the input box, the `ngModel` directive updates the value of the product price. The new price is directly reflected in the template because we use Angular interpolation syntax to display its value.

In our case, updating the current product price while entering a new one is a bad user experience. The user should be able to view the current price of the product at all times. We will modify the product detail component so that the price is displayed correctly:

1. Open the `product-detail.component.ts` file and create a `price` property inside the `ProductDetailComponent` class:

```
price: number | undefined;
```

2. Modify the `changePrice` method to use the `price` component property:

```
changePrice(product: Product) {
  this.productService.updateProduct(
    product.id,
    this.price!
  ).subscribe(() => this.router.navigate(['/products']));
}
```

3. Open the `product-detail.component.html` file and replace the binding in the `<input>` element to use the new component property:

```
<input placeholder="New price" type="number" name="price"
[(ngModel)]="price" />
```

If we run the application and try to enter a new price inside the **New price** input box, we will notice that the current price displayed does not change. The functionality of changing the price also works correctly as before.

We have seen how template-driven forms can be useful when creating small and simple forms. In the next section, we dive deeper into the alternative approach offered by the Angular framework: reactive forms.

Building reactive forms

Reactive forms, as the name implies, reactively provide access to web forms. They are built with reactivity in mind, where input controls and their values can be manipulated using observable streams. They also maintain an immutable state of form data, making them easier to test because we can be sure that the form state can be modified explicitly and consistently.

Reactive forms have a programmatic approach to creating form elements and setting up validation rules by setting everything up in the component class. The Angular key classes involved in this approach are the following:

- `FormControl`: Represents an individual form control, such as an `<input>` element.
- `FormGroup`: Represents a collection of form controls. The `<form>` element is the topmost `FormGroup` in the hierarchy of a reactive form.
- `FormArray`: Represents a collection of form controls, just like `FormGroup`, but can be modified at runtime. For example, we can add or remove `FormControl` objects dynamically as needed.

The preceding classes are available from the `@angular/forms` npm package and contain properties that can be used in the following scenarios:

- To render the UI differently according to the status of a form or control
- To check if we have interacted with a form or control

We will explore each form class through an example in our Angular application. In the following section, we will introduce reactive forms in our application using the product create component.

Interacting with reactive forms

The Angular application we have built contains a component to add new products. The component uses template reference variables to collect input data. We will use the Angular forms API to accomplish the same task using reactive forms:

1. Open the `product-create.component.ts` file and add the following import statement:

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

2. Add the `ReactiveFormsModule` class in the `imports` array of the `@Component` decorator:

```
@Component({  
  selector: 'app-product-create',
```

```
imports: [ReactiveFormsModule],  
templateUrl: './product-create.component.html',  
styleUrl: './product-create.component.css'  
})
```

The Angular forms library provides the `ReactiveFormsModule` class to create reactive forms in an Angular application.

3. Define the following `productForm` property in the `ProductCreateComponent` class:

```
productForm = new FormGroup({  
    title: new FormControl('', { nonNullable: true }),  
    price: new FormControl<number | undefined>(undefined, {  
        nonNullable: true }),  
    category: new FormControl('', { nonNullable: true })  
});
```

The `FormGroup` constructor accepts an object that contains key-value pairs of form controls. The key is a unique control name, and the value is a `FormControl` instance. The `FormControl` constructor accepts the default value of the control in the first parameter. For the `title` and the `category` controls, we pass an empty string so that we do not set any value initially. For the `price` control, which should accept numbers as values, we set it initially to `undefined`. The second parameter passed in the `FormControl` is an object that sets the `nonNullable` property to indicate that the control does not accept null values.

4. After we have created the form group and its controls, we need to associate them with the respective HTML elements in the template. Open the `product-create.component.html` file and surround the `<input>`, `<select>`, and `<button>` HTML elements with the following `<form>` element:

```
<form [formGroup]="productForm">  
    <div>  
        <label for="title">Title</label>  
        <input id="title" #title />  
    </div>  
    <div>  
        <label for="price">Price</label>  
        <input id="price" #price type="number" />  
    </div>  
    <div>
```

```
<label for="category">Category</label>
<select id="category" #category>
  <option>Select a category</option>
  <option value="electronics">Electronics</option>
  <option value="jewelery">Jewelery</option>
  <option>Other</option>
</select>
</div>
<div>
  <button (click)="createProduct(title.value, price.value,
category.value)">Create</button>
</div>
</form>
```

In the preceding template, we use the `formGroup` directive, exported from the `ReactiveFormsModule` class, to connect a `FormGroup` instance to a `<form>` element.

5. The `ReactiveFormsModule` class also exports the `formControlName` directive, which we use to connect a `FormControl` instance to an HTML element. Modify the form HTML elements as follows:

```
<div>
  <label for="title">Title</label>
  <input id="title" formControlName="title" />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" />
</div>
<div>
  <label for="category">Category</label>
  <select id="category" formControlName="category">
    <option>Select a category</option>
    <option value="electronics">Electronics</option>
    <option value="jewelery">Jewelery</option>
    <option>Other</option>
  </select>
</div>
```

In the preceding snippet, we set the value of the `formControlName` directive to the name of the respective `FormControl` instance. We also remove the template reference variables because we can get their values directly from the `FormGroup` instance.

6. Modify the `createProduct` method in the `product-create.component.ts` file accordingly:

```
createProduct() {  
    this.productsService.addProduct(this.productForm.value).  
    subscribe(() => {  
        this.router.navigate(['/products']);  
    });  
}
```

In the preceding method, we use the `value` property of the `FormGroup` class to get the form value.



Note that the `value` property does not include values from disabled fields of a form. Instead, we can use the `getRawValue` method to return values from all fields.

In this case, we can use the form value because the form model is identical to the `Product` interface.

If it was different, we could use the `controls` property of the `FormGroup` class to get form control values individually as follows:

```
createProduct() {  
    this.productsService.addProduct({  
        title: this.productForm.controls.title.value,  
        price: this.productForm.controls.price.value,  
        category: this.productForm.controls.category.value  
    }).subscribe(() => {  
        this.router.navigate(['/products']);  
    });  
}
```

The `FormControl` class contains a `value` property that returns the value of a form control.

7. Modify the `<form>` element in the `product-create.component.html` file so that we create a new product upon form submission:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
  <div>
    <label for="title">Title</label>
    <input id="title" formControlName="title" />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" type="number" />
  </div>
  <div>
    <label for="category">Category</label>
    <select id="category" formControlName="category">
      <option>Select a category</option>
      <option value="electronics">Electronics</option>
      <option value="jewelery">Jewelery</option>
      <option>Other</option>
    </select>
  </div>
  <div>
    <button type="submit">Create</button>
  </div>
</form>
```

8. Open the `global.styles.css` file and add the following CSS style:

```
label {
  margin-bottom: 4px;
  display: block;
}
```

We want the preceding styles to be available globally because we will use them in the cart component later in the chapter.

9. Open the `product-create.component.css` file and remove the style for the `<label>` tag.

If we run the application, we will see that the functionality of adding a new product still works as expected.

We learned that the `FormGroup` class groups a collection of form controls. A form control can be a single form control or another form group, as we will see in the following section.

Creating nesting form hierarchies

The product create component consists of a single form group with three form controls. Some use cases in enterprise applications require more advanced forms that involve creating nested hierarchies of form groups. Consider the following form, which is used to add a new product along with additional details:

Add new product

Title

Price

Category

▼

Additional details

Description

Photo URL

Create

Figure 10.3: New product form with additional information

The preceding form may look like a single form group, but if we take a deeper look at the component class, we will see that the `productForm` consists of two `FormGroup` instances, one nested inside the other:

```
productForm = new FormGroup({
```

```
title: new FormControl('', { nonNullable: true }),  
price: new FormControl<number | undefined>(undefined, { nonNullable:  
true }),  
category: new FormControl('', { nonNullable: true }),  
extra: new FormGroup({  
  image: new FormControl(''),  
  description: new FormControl('')  
})  
});
```

The `productForm` property is the parent form group, while `extra` is its child. A parent form group can have as many children form groups as it needs. If we take a look at the component template, we will see that the child form group is defined differently from the parent one:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">  
  <div>  
    <label for="title">Title</label>  
    <input id="title" formControlName="title" />  
  </div>  
  <div>  
    <label for="price">Price</label>  
    <input id="price" formControlName="price" type="number" />  
  </div>  
  <div>  
    <label for="category">Category</label>  
    <select id="category" formControlName="category">  
      <option>Select a category</option>  
      <option value="electronics">Electronics</option>  
      <option value="jewelery">Jewelery</option>  
      <option>Other</option>  
    </select>  
  </div>  
  <h2>Additional details</h2>  
  <form formGroupName="extra">  
    <div>  
      <label for="descr">Description</label>  
      <input id="descr" formControlName="description" />  
    </div>
```

```
<div>
  <label for="photo">Photo URL</label>
  <input id="photo" formControlName="image" />
</div>
</form>
<div>
  <button type="submit">Create</button>
</div>
</form>
```

In the preceding HTML template, we use the `formGroupName` directive to bind the inner form element to the `extra` property.



You may have expected to bind it directly to the `productForm.extra` property, but Angular is pretty smart because it understands that `extra` is a child form group of `productForm`. It can deduce this information because the form element related to `extra` is inside the form element that binds to the `productForm` property.

The value of a child form group is shared with its parent in a nested form hierarchy. In our case, the value of the `extra` form group will be included in the `productForm` group, thereby maintaining a consistent form model.

We have already covered the `FormGroup` and `FormControl` classes. In the following section, we will learn how to use the `FormArray` class to interact with dynamic forms.

Modifying forms dynamically

Consider the scenario where we have added some products to the shopping cart of our e-shop application and want to update their quantities before checking out the order.

Currently, our application does not have any functionality for a shopping cart, so we will now add one:

1. Run the following command to create a `Cart` interface:

```
ng generate interface Cart
```

2. Open the `cart.ts` file and modify the `Cart` interface as follows:

```
export interface Cart {
  id: number;
```

```
    products: { productId :number }[];  
}
```

In the preceding snippet, the `products` property will contain the product IDs that belong to the current cart.

3. Create a new service to manage the shopping cart by running the following Angular CLI command:

```
ng generate service cart
```

4. Open the `cart.service.ts` file and modify the import statements as follows:

```
import { Injectable, inject } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
import { Observable, defer, map } from 'rxjs';  
import { Cart } from './cart';  
import { APP_SETTINGS } from './app.settings';
```

5. Create the following properties in the `CartService` class:

```
cart: Cart | undefined;  
private cartUrl = inject(APP_SETTINGS).apiUrl + '/carts';
```

The `cartUrl` property is used for the cart endpoint of the Fake Store API and the `cart` property to keep a local cache of the user cart.

6. Inject the `HttpClient` service in the constructor:

```
constructor(private http: HttpClient) { }
```

7. Add the following method to add a product to the cart:

```
addProduct(id: number): Observable<Cart> {  
  const cartProduct = { productId: id, quantity: 1 };  
  
  return defer(() =>  
    !this.cart  
    ? this.http.post<Cart>(this.cartUrl, { products: [cartProduct] })  
    : this.http.put<Cart>(`${this.cartUrl}/${this.cart.id}`, {  
      products: [  
        ...this.cart.products,  
        cartProduct  
      ]  
    })  
  );  
}
```

```

        cartProduct
    ]
})
).pipe(map(cart => this.cart = cart));
}

```

In the preceding method, we use a new RxJS operator called `defer`. The `defer` operator works as an if/else statement for observables.

If the `cart` property has not been initialized, which means that our cart is currently empty, we initiate a POST request to the API passing the `cartProduct` variable as a parameter. Otherwise, we initiate a PATCH request passing the `cartProduct` along with the existing products from the cart.

We have completed the setup of our service so that it can communicate with the Fake Store API. Now, we need to connect the service with the respective component:

1. Open the `product-detail.component.ts` file and add the following `import` statement:

```
import { CartService } from '../cart.service';
```

2. Inject `CartService` in the `ProductDetailComponent` class:

```

constructor(
  private productService: ProductsService,
  public authService: AuthService,
  private route: ActivatedRoute,
  private router: Router,
  private cartService: CartService
) { }

```

3. Modify the `addToCart` method so that it calls the `addProduct` method of the `CartService` class:

```

addToCart(id: number) {
  this.cartService.addProduct(id).subscribe();
}

```

4. Finally, open the `product-detail.component.html` file and modify the `click` event of the `Add to cart` button:

```
<button (click)="addToCart(product.id)">Add to cart</button>
```

We have implemented the basic functionality for storing the selected products that users want to buy. Now, we must modify the cart component to display the cart items:

1. Open the `cart.component.ts` file and modify the `import` statements as follows:

```
import { Component, OnInit } from '@angular/core';
import {
  FormArray,
  FormControl,
  FormGroup,
  ReactiveFormsModule
} from '@angular/forms';
import { Product } from '../product';
import { CartService } from '../cart.service';
import { ProductsService } from '../products.service';
```

2. Add the `ReactiveFormsModule` class in the `imports` array of the `@Component` decorator:

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

3. Add the `OnInit` interface to the list of implemented interfaces of the `CartComponent` class:

```
export class CartComponent implements OnInit
```

4. Create the following properties in the TypeScript class:

```
cartForm = new FormGroup({
  products: new FormArray<FormControl<number>>([])
});
products: Product[] = [];
```

In the preceding snippet, we created a `FormGroup` object containing a `products` property. We set the value of the `products` property to an instance of the `FormArray` class. The constructor of the `FormArray` class accepts a list of `FormControl` instances with the type `number` as a parameter. The list is empty for now since the cart has no products. The `products` property outside the `FormGroup` instance will be used for lookup reasons to display the title of each product in the cart.

5. Add a constructor to inject the following services:

```
constructor(  
    private cartService: CartService,  
    private productService: ProductsService  
) {}
```

6. Create the following method to get products from the cart:

```
private getProducts() {  
    this.productService.getProducts().subscribe(products => {  
        this.cartService.cart?.products.forEach(item => {  
            const product = products.find(p => p.id === item.productId);  
            if (product) {  
                this.products.push(product);  
            }  
        });  
    });  
}
```

In the preceding method, we initially subscribe to the `getProducts` method of the `ProductsService` class to get the available products. Then, for each product in the cart, we extract the `productId` property and check if it exists inside the cart. If the product is found, we add it to the `products` component property.

7. Create another method to build our form:

```
private buildForm() {  
    this.products.forEach(() => {  
        this.cartForm.controls.products.push(  
            new FormControl(1, { nonNullable: true })  
        );  
    });  
}
```

In the preceding method, we iterate over the `products` property and add a `FormControl` instance for each one inside the `products` form array. We set the value of each form control to 1 to indicate that the cart contains one item of each product by default.

8. Create the following `ngOnInit` method that combines both methods from steps 6 and 7:

```
ngOnInit(): void {
  this.getProducts();
  this.buildForm();
}
```

9. Open the `cart.component.html` file and replace its HTML template with the following content:

```
<div [formGroup]="cartForm">
  <div formArrayName="products">
    @for(product of cartForm.controls.products.controls; track
    $index) {
      <label>{{products[$index].title}}</label>
      <input [formControlName]="$index" type="number" />
    }
  </div>
</div>
```

In the preceding template, we use a `@for` block to iterate over the `controls` property of the `products` form array and create an `<input>` element for each one. We use the `$index` keyword of the `@for` block to give a dynamically created name to each form control using the `formControlName` binding. We have also added a `<label>` tag that displays the product title from the `products` component property. The product title is fetched using the `$index` of the current product in the array.

10. Finally, open the `cart.component.css` files and add the following CSS styles:

```
:host {
  width: 500px;
}

input {
  width: 50px;
}
```

To see the `cart` component in action, run the application using the `ng serve` command and add some products to the cart.



Do not forget to log in first because the functionality that adds a product to the cart is available only to authenticated users.

After adding some products to the cart, click the **My Cart** link to view your shopping cart. It should look like the following:

SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s

1

Mens Cotton Jacket

1

WD 2TB Elements Portable External Hard Drive - USB 3.0

1

Figure 10.4: Shopping cart

Since we have established the business logic for managing a shopping cart, we can also update the checkout guard we created in the previous chapter:

1. Open the `checkout.guard.ts` file and add the following `import` statements:

```
import { inject } from '@angular/core';
import { CartService } from './cart.service';
```

2. Inject the `CartService` class in the `checkoutGuard` function using the following statement:

```
const cartService = inject(CartService);
```

3. Modify the remaining body of the `checkoutGuard` arrow function so that we display the confirmation dialog only when the cart is not empty:

```
if (cartService.cart) {
  const confirmation = confirm(
    'You have pending items in your cart. Do you want to continue?'
  );
  return confirmation;
}
return true;
```

With the `FormArray`, we have completed our exploration of the most basic building blocks of an Angular form. We learned how to use Angular forms classes to create structured web forms and collect user input. In the following section, we will learn how to build Angular forms using the `FormBuilder` service.

Using a form builder

Using form classes to build Angular forms can become repetitive and tedious for complex scenarios. The Angular framework provides `FormBuilder`, a built-in service to Angular forms that contains helper methods for building forms. Let's see how we could use it to build a form for creating new products:

1. Open the `product-create.component.ts` file and import the `OnInit` and `FormBuilder` artifacts:

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

2. Add `OnInit` to the list of implemented interfaces in the `ProductCreateComponent` class:

```
export class ProductCreateComponent implements OnInit
```

3. Inject the `FormBuilder` class in the constructor:

```
constructor(
  private productService: ProductsService,
  private router: Router,
  private builder: FormBuilder
) {}
```

4. Modify the `productForm` property as follows:

```
productForm: FormGroup<{
  title: FormControl<string>,
  price: FormControl<number | undefined>,
  category: FormControl<string>
}> | undefined;
```

In the preceding snippet, we define only the structure of the form because it will now be created using the `FormBuilder` service.

5. Create the following method to build the form:

```
private buildForm() {  
    this.productForm = this.builder.nonNullable.group({  
        title: ['',  
        price: this.builder.nonNullable.control<number |  
undefined>(undefined),  
        category: ['']  
    });  
}
```

In the preceding method, we use the `nonNullable` property of the `FormBuilder` class to create a form group that cannot be null. The `group` method is used to group form controls. The `title` and `category` form controls are created using an empty string as the default value. The `price` form control follows a different approach from the rest because we cannot assign a default value of `undefined` due to TypeScript language limitations. In this case, we use the `control` method of the `nonNullable` property to define the form control.

6. Add the `ngOnInit` lifecycle hook to execute the `buildForm` method:

```
ngOnInit(): void {  
    this.buildForm();  
}
```

7. Add the non-null assertion operator when accessing the `productForm` property in the `createProduct` method:

```
createProduct() {  
    this.productsService.addProduct(this.productForm!.value).  
subscribe(() => {  
    this.router.navigate(['/products']);  
});  
}
```

8. Open the `product-create.component.html` file and add the non-null assertion operator in the `<form>` HTML element also:

```
<form [formGroup]="productForm!" (ngSubmit)="createProduct()">  
    <div>  
        <label for="title">Title</label>
```

```
<input id="title" formControlName="title" />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" />
</div>
<div>
  <label for="category">Category</label>
  <select id="category" formControlName="category">
    <option>Select a category</option>
    <option value="electronics">Electronics</option>
    <option value="jewelery">Jewelery</option>
    <option>Other</option>
  </select>
</div>
<div>
  <button type="submit">Create</button>
</div>
</form>
```

Using the `FormBuilder` service to create Angular forms, we don't have to deal with the `FormGroup` and `FormControl` data types explicitly, although that is what is being created under the hood.

Run the application using the `ng serve` command and verify that the new product creation process works correctly. Try to click the **Create** button without entering any values in the form controls and observe what happens in the product list. The application creates a product with an empty title. It is a situation that we should avoid in a real-world scenario. We should be aware of the status of a form control and take action accordingly.



The example code in the rest of the chapter does not use the `FormBuilder` service when working with reactive forms.

In the following section, we'll investigate different properties that we can check to get form status and provide feedback to the user.

Validating input in forms

An Angular form should validate input and provide visual feedback to enhance UX and guide users to complete the form successfully. We will investigate the following ways to validate forms in Angular applications:

- Global validation with CSS
- Validation in the component class
- Validation in the component template
- Building custom validators

In the following section, we will learn how to apply validation rules globally in an Angular application using CSS styles.

Global validation with CSS

The Angular framework sets the following CSS classes automatically in a form, template-driven or reactive, that we can use to provide user feedback:

- `ng-untouched`: Indicates that we have not interacted with a form yet
- `ng-touched`: Indicates that we have interacted with a form
- `ng-dirty`: Indicates that we have set a value to a form
- `ng-pristine`: Indicates that we have not modified a form yet

Furthermore, Angular adds the following classes on the HTML element of a form control:

- `ng-valid`: Indicates that the value of a form is valid
- `ng-invalid`: Indicates that the value of a form is not valid

Angular sets the preceding CSS classes in the form and its controls according to their status. The form status is evaluated according to the status of its controls. For example, if at least one form control is invalid, Angular will set the `ng-invalid` CSS class to the form and the corresponding control.



In the case of nested form hierarchies, the status of a child form group is bubbled up to the hierarchy and shared with its parent form.

We can use the built-in CSS classes and style Angular forms using CSS only. For example, to display a light blue highlighted border in an input control when interacting with that control for the first time, we should add the following style:

```
input.ng-touched {  
    border: 3px solid lightblue;  
}
```

We can also combine CSS classes according to the needs of our application:

1. Open the global `styles.css` file and modify the `input.valid` style as follows:

```
input.valid, input.ng-dirty.ng-valid {  
    border: solid green;  
}
```

The preceding style will display a green border when an input control has a valid value entered by the user.

2. Modify the `input.invalid` style accordingly:

```
input.invalid, input.ng-dirty.ng-invalid {  
    border: solid red;  
}
```

The preceding style will display a red border when an input control has an invalid value entered by the user.

3. Open the `product-create.component.html` file and add the required attribute in the `<input>` form controls:

```
<div>  
    <label for="title">Title</label>  
    <input id="title" formControlName="title" required />  
</div>  
  
<div>  
    <label for="price">Price</label>  
    <input id="price" formControlName="price" type="number" required  
/>  
</div>
```

4. Run the application using the `ng serve` command and navigate to `http://localhost:4200/products/new`.
5. Enter some text into the **Title** field and click outside of the input control. Notice that it has a green border.
6. Remove the text from the **Title** field and click outside of the input control. The border should now turn red.

We learned how to define validation rules in the template using CSS styles. In the following section, we will learn how to define them in template-driven forms and give visual feedback using appropriate messages.

Validation in template-driven forms

In the preceding section, we learned that Angular adds a collection of built-in CSS classes while validating Angular forms. Each class has a corresponding boolean property in the respective form model, both in template-drive and reactive forms:

- `untouched`: Indicates that we have not interacted with a form yet
- `touched`: Indicates that we have interacted with a form
- `dirty`: Indicates that we have set a value to a form
- `pristine`: Indicates that we have not modified a form yet
- `valid`: Indicates that the value of a form is valid
- `invalid`: Indicates that the value of a form is not valid

We can leverage the preceding classes and inform the user about the current form status. First, let's investigate the behavior of the change price process in the product details component:

1. Run the `ng serve` command to start the application and navigate to `http://localhost:4200`.
2. Select a product from the list.
3. Add a value of `0` into the **New price** input box and click the **Change** button.
4. Select the same product from the list and observe the output:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€0.00

New price

Change

electronics

Delete

Figure 10.5: Product details

The presentation logic of the component fails to detect that the user can enter 0 for the product price. A product should always have a price.

The product details component needs to validate the input of the price value, and if the input is found to be invalid, disable the **Change** button, and display an informational message to the user.



Handling validation is a matter of personal preference or business specification. In this scenario, we decided to showcase a common validation approach by disabling the button and displaying an appropriate message.

Template-driven validation is performed in the component template. Open the `product-detail.component.html` file and execute the following steps:

1. Create the `priceCtrl` template reference variable and bind it to the `ngModel` property:

```
<input  
  placeholder="New price"  
  type="number"  
  name="price"  
  #priceCtrl="ngModel"  
  [(ngModel)]="price" />
```

The `ngModel` property gives us access to the underlying form control model.

2. Add the `required` and `min` validation attributes to the `<input>` element:

```
<input  
  placeholder="New price"  
  type="number"  
  name="price"  
  required min="1"  
  #priceCtrl="ngModel"  
  [(ngModel)]="price" />
```

The `min` validation attribute can be used only with `<input>` HTML elements of the `number` type. It is used to define the minimum value when using the arrows of the numeric control.

3. Add the following `` HTML element underneath the `<button>` element of the form:

```
@if (priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.  
hasError('min')))  
<span class="help-text">Please enter a valid price</span>  
}
```

The preceding HTML element will be displayed when we enter a price value and then either leave it blank or enter a zero. We use the `hasError` method of the form control model to check if the `min` validation throws an error.



All validation attributes can be checked using the `hasError` method. The validity status of a control is evaluated based on the status of all validation attributes we attach to the HTML element.

4. Add a `priceForm` template reference variable in the `<form>` HTML element and bind it to the `ngForm` property:

```
<form (ngSubmit)="changePrice(product)" #priceForm="ngForm">  
<input  
  placeholder="New price"  
  type="number"  
  name="price"  
  required min="1"  
  #priceCtrl="ngModel"  
  [(ngModel)]="price" />
```

```
<button class="secondary" type="submit">Change</button>
@if (priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.
hasError('min'))) {
    <span class="help-text">Please enter a valid price</span>
}
</form>
```

The `ngForm` property gives us access to the underlying form model.

- Bind the `disabled` property of the `<button>` HTML element to the `invalid` status of the form model:

```
<button
    class="secondary"
    type="submit"
    [disabled]="priceForm.invalid">
    Change
</button>
```



In the preceding template, we could bind directly to the `priceCtrl.invalid` status since the form has only one control. We choose the form instead for demonstration purposes.

- Open the `styles.css` file and add the following CSS styles for the `` tag and the `disabled` button:

```
.help-text {
    display: flex;
    color: var(--hot-red);
    font-size: 0.875rem;
}

button:disabled {
    background-color: lightgrey;
    cursor: not-allowed;
}
```

To verify that the validation works as intended, execute the following steps:

- Run the `ng serve` command to start the application and select a product from the list.

2. Enter 0 in the **New price** input box and observe the output:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III
6 Gb/s**

€109.00

The screenshot shows a user interface for a product creation form. At the top, the product details are listed: "You selected: SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s" and "€109.00". Below this, there is a form group with a red border around the "New price" input field. The input field contains the value "0". To the right of the input field is a "Change" button. A red error message "Please enter a valid price" is displayed below the input field. Further down the page, there is a category selection "electronics" and a "Delete" button.

Figure 10.6: Validation error

3. Enter a valid value and verify that the error message is gone and the **Change** button is enabled.
4. Leave the **New price** input box blank and verify that the error message is displayed again and the **Change** button is disabled.

Now that we have learned how to accomplish validation in template-driven forms, let's see how to validate input data in reactive ones.

Validation in reactive forms

Template-driven forms rely solely on the component template to perform validations. In reactive forms, the source of truth is our form model that resides in the TypeScript class of the component. We define validation rules in reactive forms when building the `FormGroup` instance programmatically.

To demonstrate validation in reactive forms, we will add validation rules in the product create component:

1. Open the `product-create.component.ts` file and import the `Validators` class from the `@angular/forms` npm package:

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

2. Modify the declaration of the `productForm` property so that the `title` and `price` form controls pass a `validators` property in the `FormControl` instance:

```
productForm = new FormGroup({
  title: new FormControl('', {
    nonNullable: true,
    validators: Validators.required
}),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true,
    validators: [Validators.required, Validators.min(1)]
}),
  category: new FormControl('', { nonNullable: true })
});
```

The `Validators` class contains a static field for each validation rule available. It contains almost the same validator rules that are available for template-driven forms. We can combine multiple validators by adding them to an array, as indicated by the `validators` property in the `price` form control.



When we add a validator using the `FormControl` class, we can remove the respective HTML attribute from the HTML template. However, it is recommended to keep it for accessibility purposes so that screen-reader applications can use it.

3. Open the `product-create.component.html` file and use the `invalid` property of the `productForm` property to disable the Create button:

```
<button type="submit" [disabled]="productForm.invalid">Create</button>
```

4. Add a `` HTML element in each `<input>` form control to display an error message when the control has been touched, and the required validation throws an error:

```
<div>
  <label for="title">Title</label>
  <input id="title" formControlName="title" required />
  @if (productForm.controls.title.touched && productForm.controls.
  title.invalid) {
    <span class="help-text">Title is required</span>
  }
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required
  />
  @if (productForm.controls.price.touched && productForm.controls.
  price.invalid) {
    <span class="help-text">Price is required</span>
  }
</div>
```

In the preceding snippet, we use the `controls` property of the `productForm` property to get access to the individual form control models and get their statuses.

5. It would be nice to display different messages depending on the validation rule. We could display a more specific message when the `min` validation of the `price` control throws an error, for example. We can use the `hasError` method that we saw in the preceding section to display such a message:

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required
  />
  @if (productForm.controls.price.touched && productForm.controls.
  price.hasError('required')) {
    <span class="help-text">Price is required</span>
  }
  @if (productForm.controls.price.touched && productForm.controls.
  price.hasError('min')) {
    <span class="help-text">Price should be greater than 0</span>
```

```
    }  
</div>
```

The Angular framework provides a set of built-in validators that we learned to use in our forms. In the following section, we will learn how to create a custom validator for template-driven and reactive forms to satisfy particular business needs.

Building custom validators

Built-in validators won't cover all the scenarios we might encounter in an Angular application; however, writing a custom validator and using it in an Angular form is easy. In our case, we will build a validator to check that the price of a product cannot exceed a specified threshold.



We could use the built-in `max` validator to accomplish the same task. However, we will be building the validator function for learning purposes.

Custom validators are used when we want to validate a form or a control with custom code. For example, to communicate with an API for validating a value, or to perform a complex calculation for validating a value.

1. Create a file named `price-maximum.validator.ts` in the `src\app` folder and add the following contents:

```
import { ValidatorFn, AbstractControl, ValidationErrors } from '@  
angular/forms';  
  
export function priceMaximumValidator(price: number): ValidatorFn {  
  return (control: AbstractControl): ValidationErrors | null => {  
    const isMax = control.value <= price;  
    return isMax ? null : { priceMaximum: true };  
  };  
}
```

A form validator is a function that returns a `ValidationErrors` object with the error specified or a `null` value. It accepts the form control to which it will be applied as a parameter. In the preceding snippet, if the control value is larger than a specific threshold passed in the `price` parameter of the exported function, it returns a validation error object. Otherwise, it returns `null`.

The key of the validation error object specifies a descriptive name for the validator error. It is a name we can later check with the `hasError` method of the control to find out if it has any errors. The value of the validation error object can be any arbitrary value that we can pass in the error message.

2. Open the `product-create.component.ts` file and add the following `import` statement:

```
import { priceMaximumValidator } from '../price-maximum.validator';
```

3. Add the validator in the `validators` array of the `price` form control and set the threshold to 1000:

```
price: new FormControl<number | undefined>(undefined, {
  nonNullable: true,
  validators: [
    Validators.required,
    Validators.min(1),
    priceMaximumValidator(1000)
  ]
})
```

4. Add a new `` HTML element for the `price` form control in the `product-create.component.html` file:

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" type="number" required />
  @if (productForm.controls.price.touched && productForm.controls.price.hasError('required')) {
    <span class="help-text">Price is required</span>
  }
  @if (productForm.controls.price.touched && productForm.controls.price.hasError('min')) {
    <span class="help-text">Price should be greater than 0</span>
  }
  @if (productForm.controls.price.touched && productForm.controls.price.hasError('priceMaximum')) {
    <span class="help-text">Price must be smaller or equal to 1000</span>
  }
</div>
```

5. Run the `ng serve` command to start the application and navigate to `http://localhost:4200/products/new`.
6. Enter a value of `1200` in the **Price** field, click outside the input box, and observe the output:

Add new product

Title

Price

 1200

Price must be smaller or equal to 1000

Category

 ▼

Create

Figure 10.7: Validation in reactive forms

To use the price maximum validator in a template-driven form, we must follow a different approach that involves creating an Angular directive:

1. Run the following command to create an Angular directive:

```
ng generate directive price-maximum
```

The preceding directive will act as a wrapper over the `priceMaximumValidator` function we have already created.

2. Open the `price-maximum.directive.ts` file and modify the `import` statements as follows:

```
import { Directive, Input, NumberAttribute } from '@angular/core';
import { AbstractControl, NG_VALIDATORS, ValidationErrors, Validator }
} from '@angular/forms';
import { priceMaximumValidator } from './price-maximum.validator';
```

3. Add the `NG_VALIDATORS` provider in the `@Directive` decorator:

```
@Directive({
  selector: '[appPriceMaximum]',
```

```
providers: [
  {
    provide: NG_VALIDATORS,
    useExisting: PriceMaximumDirective,
    multi: true
  }
]
})
```

The `NG_VALIDATORS` token is a built-in token of Angular forms that helps us register an Angular directive as a form validator. In the preceding snippet, we use the `multi` property in the provider configuration because we can register multiple directives with the `NG_VALIDATORS` token.

4. Add the `Validator` interface in the implemented interfaces of the `PriceMaximumDirective` class:

```
export class PriceMaximumDirective implements Validator
```

5. Add the following input property that will be used to pass a value for the maximum threshold:

```
appPriceMaximum = input(undefined, {
  alias: 'threshold',
  transform: numberAttribute
});
```

In the preceding property, we pass a configuration object with two properties as a parameter in the `input` function. The `alias` property defines the name of the input property that we will use for binding. The `transform` property is used to convert the value of the input property to a different type. The `numberAttribute` is a built-in function of the Angular framework that converts the input property value to a number.



Angular also contains the `booleanAttribute` function, which parses an input property value as a boolean.

6. Implement the `validate` method of the `Validator` interface as follows:

```
validate(control: AbstractControl): ValidationErrors | null {
```

```
    return this.appPriceMaximum
      ? priceMaximumValidator(this.appPriceMaximum()!)(control)
      : null;
}
```

The signature of the validate method is the same as the function returned from the priceMaximumValidator function. It checks the appPriceMaximum input property and accordingly delegates its value to the priceMaximumValidator function.

We will use the new directive we created in the product detail component:

1. Open the `product-detail.component.ts` file and add the following import statement:

```
import { PriceMaximumDirective } from '../price-maximum.directive';
```

2. Add the `PriceMaximumDirective` class in the imports array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-detail',
  imports: [
    CommonModule,
    FormsModule,
    PriceMaximumDirective
  ],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css']
})
```

3. Open the `product-detail.component.html` file and add the new validator in the `<input>` HTML element:

```
<input
  placeholder="New price"
  type="number"
  name="price"
  required min="1"
  appPriceMaximum threshold="500"
  #priceCtrl="ngModel"
  [(ngModel)]="price" />
```

4. Add a new `` HTML element to display a different message when the validator throws an error:

```
@if (priceCtrl.dirty && priceCtrl.hasError('priceMaximum')) {  
    <span class="help-text">Price must be smaller or equal to 500</span>  
}
```

5. Run the `ng serve` command to start the application and select a product from the list.
6. Enter the value `600` in the **New price** input box and observe the output:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€109.00

A screenshot of an Angular form. At the top, it displays the selected product: "SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s". Below this, the current price is shown as "€109.00". A red-bordered input field contains the value "600". To the right of the input field is a "Change" button. Below the input field, two error messages are displayed in red: "Please enter a valid price" and "Price must be smaller or equal to 500". Further down, there is a grey button labeled "electronics" and a red button labeled "Delete".

Figure 10.8: Validation in template-driven forms

Angular custom validations can work synchronously or asynchronously. In this section, we learned how to work with the former. Asynchronous validations are an advanced topic that we will not cover in this book. However, you can learn more at <https://angular.dev/guide/forms/form-validation#creating-asynchronousValidators>.

In the following section, we will explore manipulating the state of an Angular form.

Manipulating form state

The state of an Angular form differs between template-driven and reactive forms. In the former, the state is a plain object, whereas in the latter, it is kept in the form model. In this section, we will learn about the following concepts:

- Updating form state
- Reacting to state changes

We will start by exploring how we can change the form state.

Updating form state

Working with the form state in template-driven forms is relatively easy. We must interact with the component property bound to the `ngModel` directive of a form control.

In reactive forms, we can use the `value` property of a `FormControl` instance or the following methods of the `FormGroup` class to change values in the whole form:

- `setValue`: Replaces values in all controls of the form
- `patchValue`: Updates values in specific controls of the form

The `setValue` method accepts an object as a parameter that contains key-value pairs for all form controls. If we want to fill in the details of a product in the product create component programmatically, the following snippet serves as an example:

```
this.productForm.setValue({  
  title: 'TV monitor',  
  price: 600,  
  category: 'electronics'  
});
```

In the preceding snippet, each key of the object passed in the `setValue` method must match the name of each form control. If we omit one, Angular will throw an error.

If we want to fill in some of the details of a product, we can use the `patchValue` method:

```
this.productForm.patchValue({  
  title: 'TV monitor',  
  category: 'electronics'  
});
```

The `setValue` and `patchValue` methods of the `FormGroup` class help us set data in a form.

Another interesting aspect of forms is that we can be notified when these values change, as we will see in the following section.

Reacting to state changes

A common scenario when working with Angular forms is that we want to trigger a side effect when the value of a form control changes. A side effect can be any of the following:

- To alter the value of a form control
- To initiate an HTTP request to filter the value of a form control
- To enable/disable certain parts of the component template

In template-driven forms, we can use an extended version of the `ngModel` directive to get notified when its value changes. The `ngModel` directive contains the following bindable properties:

- `ngModel`: An input property for passing values to the control
- `ngModelChange`: An output property for getting notified when the control value changes

We can write the `ngModel` binding in the `<input>` HTML element of the product detail component in the following alternate way:

```
<input  
  placeholder="New price"  
  type="number"  
  name="price"  
  required min="1"  
  appPriceMaximum threshold="500"  
  #priceCtrl="ngModel"  
  [ngModel]="price"  
  (ngModelChange)="price = $event" />
```

In the preceding snippet, we set the value of the `ngModel` input property using property binding and the value of the `price` component property using event binding. Angular triggers the `ngModelChange` event automatically and includes the new value of the `<input>` HTML element in the `$event` property. We can use the `ngModelChange` event for any side effects in our component when the value of the `price` form control changes.

In reactive forms, we use an observable-based API to react to state changes. The `FormGroup` and `FormControl` classes contain the `valueChanges` observable, which we can use to subscribe and get notified when the value of the form or control changes.

We will use it to reset the value of the `price` form control in the product create component when the category changes:

1. Open the `product-create.component.ts` file and import the `OnInit` artifact from the `@angular/core` npm package:

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

2. Add the `OnInit` interface to the list of the `ProductCreateComponent` class implemented interfaces:

```
export class ProductCreateComponent implements OnInit
```

3. Create the following `ngOnInit` method to subscribe to the `valueChanges` property of the category form control:

```
ngOnInit(): void {
  this.productForm.controls.category.valueChanges.subscribe(() => {
    this.productForm.controls.price.reset();
  });
}
```

In the preceding method, we reset the value of the `price` form control by using the `reset` method of the `FormControl` class.



The `valueChanges` property of the `FormControl` class is a standard observable stream. Do not forget to unsubscribe when the component is destroyed.

Of course, there is more that we can do with the `valueChanges` observable; for example, we could check if the product title is already reserved by sending it to a backend API. Hopefully, however, the preceding examples have conveyed how you can take advantage of the reactive nature of forms and respond accordingly.

Summary

In this chapter, we learned that Angular provides two different flavors for creating forms – template-driven and reactive – and neither approach is better than the other. We explored how to build each form type and perform validations on input data, and covered custom validations for implementing additional validation scenarios. We also learned how to update the state of a form and how to react when the values in the state change.

In the following chapter, we will explore various ways of handling application errors. Error handling is a very important feature of an Angular application and can have different sources and reasons, as we will see.

11

Handling Application Errors

Application errors are an integral part of the lifetime of a web application. They can occur either during runtime or while developing the application. Possible causes of a runtime error are an HTTP request that failed or an incomplete HTML form. A web application must handle runtime errors and mitigate unwanted effects to ensure a smooth user experience.

Development errors usually happen when we do not properly use a programming language or framework according to its semantics. In this case, errors may override the compiler and surface in the application while running. Development errors can be mitigated by following best practices and recommended coding techniques.

In this chapter, we will learn how to handle different types of errors in an Angular application and understand errors from the framework itself. We will explore the following concepts in more detail:

- Handling runtime errors
- Demystifying framework errors

Technical requirements

The code samples described in this chapter can be found in the ch11 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Handling runtime errors

The most common runtime errors in an Angular application come from the interaction with an HTTP API. Entering the wrong login credentials or sending data in the wrong format can result in an HTTP error. An Angular application can handle HTTP errors in the following ways:

- Explicitly during the execution of a particular HTTP request
- Globally in the global error handler of the application
- Centrally using an HTTP interceptor

In the following section, we will explore how to handle an HTTP error in a specific HTTP request.

Catching HTTP request errors

Handling errors in HTTP requests typically requires manually inspecting the information returned in the error response object. RxJS provides the `catchError` operator to simplify that. It can catch potential errors when initiating an HTTP request with the `pipe` operator.



You will need the source code of the Angular application we created in *Chapter 10, Collecting User Data with Forms*, to follow along with the rest of the chapter.

Let's see how we could use the `catchError` operator to catch HTTP errors while fetching the product list in our application:

1. Open the `products.service.ts` file and import the `catchError` and `throwError` operators from the `rxjs` npm package:

```
import { Observable, map, of, tap, catchError, throwError } from  
'rxjs';
```

2. Import the `HttpErrorResponse` interface from the `@angular/common/http` namespace:

```
import { HttpClient, HttpParams, HttpErrorResponse } from '@angular/  
common/http';
```

3. Modify the `getProducts` method accordingly:

```
getProducts(limit?: number): Observable<Product[]> {  
  if (this.products.length === 0) {  
    const options = new HttpParams().set('limit', limit || 10);  
    return this.http.get<Product[]>(this.productsUrl, {
```

```
    params: options
  }).pipe(
    map(products => {
      this.products = products;
      return products;
    }),
    catchError((error: HttpErrorResponse) => {
      console.error(error);
      return throwError(() => error);
    })
  );
}

return of(this.products);
}
```

The signature of the `catchError` operator contains the actual `HttpErrorResponse` object that is returned from the server. After catching the error, we use the `throwError` operator, which re-throws the error as an observable.



Alternatively, we could have used the `throw` keyword from the standard web API methods to throw the error. However the `throwError` method is, most of the time, overkill. Please use it accordingly.

This way, we ensure that the application execution will continue and complete without causing a potential memory leak.

In a real-world scenario, we would probably create a helper method to log the error in a more solid tracking system and return something meaningful according to the cause of the error:

1. In the same file, `products.service.ts`, import the `HttpStatus` enumeration from the `@angular/common/http` namespace:

```
import { HttpClient, HttpHeaders, HttpErrorResponse, HttpStatusCode }  
from '@angular/common/http';
```

`HttpStatusCode` is an enumeration that contains a list of all HTTP response status codes.

2. Create the following method in the `ProductsService` class:

```
private handleError(error: HttpErrorResponse) {  
  let message = '';
```

```
switch(error.status) {  
  case HttpStatusCode.InternalServerError:  
    message = 'Server error';  
    break;  
  case HttpStatusCode.BadRequest:  
    message = 'Request error';  
    break;  
  default:  
    message = 'Unknown error';  
}  
  
console.error(message, error.error);  
  
return throwError(() => error);  
}
```

The preceding method logs a different message in the browser console according to the error status. It uses a `switch` statement to differentiate between internal server errors and bad requests. For any other errors, it falls back to the `default` statement, which logs a generic message in the console.

3. Refactor the `getProducts` method to use the `handleError` method to catch errors:

```
getProducts(limit?: number): Observable<Product[]> {  
  if (this.products.length === 0) {  
    const options = new HttpParams().set('limit', limit || 10);  
    return this.http.get<Product[]>(this.productsUrl, {  
      params: options  
    }).pipe(  
      map(products => {  
        this.products = products;  
        return products;  
      }),  
      catchError(this.handleError)  
    );  
  }  
  return of(this.products);  
}
```

The `handleError` method currently manages HTTP errors originating only from the HTTP response. However, other errors can occur in an Angular application from the client side, such as a request that did not reach the server due to a network error or an exception thrown in an RxJS operator. To handle any of the previous errors, we should add a new `case` statement in the `handleError` method:

```
private handleError(error: HttpErrorResponse) {
  let message = '';

  switch(error.status) {
    case 0:
      message = 'Client error';
      break;
    case HttpStatusCode.InternalServerError:
      message = 'Server error';
      break;
    case HttpStatusCode.BadRequest:
      message = 'Request error';
      break;
    default:
      message = 'Unknown error';
  }

  console.error(message, error.error);

  return throwError(() => error);
}
```

In the preceding snippet, an error with a status of `0` indicates that it is an error that occurred on the client side of the application.

Error handling in HTTP requests could be combined with a mechanism that retries a given HTTP call a specific amount of times before handling the error. There is an RxJS operator for nearly everything, even one for retrying HTTP requests. It accepts the number of retries where the particular request has to be executed until it completes successfully:

```
getProducts(limit?: number): Observable<Product[]> {
  if (this.products.length === 0) {
    const options = new HttpParams().set('limit', limit || 10);
```

```
        return this.http.get<Product[]>(this.productsUrl, {
            params: options
        }).pipe(
            map(products => {
                this.products = products;
                return products;
            }),
            retry(2),
            catchError(this.handleError)
        );
    }
    return of(this.products);
}
```

We learned that we use the `catchError` RxJS operator to capture errors. The way we handle it depends on the scenario. In our case, we created a `handleError` method for all HTTP calls in a service. In a real-world scenario, we would follow the same approach of error handling in other Angular services of an application. Creating one method for each service would not be convenient and does not scale well.

Alternatively, we could utilize the global error handler that Angular provides to handle errors in a central place. We will learn how to create a global error handler in the next section.

Creating a global error handler

The Angular framework provides the `ErrorHandler` class for handling errors globally in an Angular application. The default implementation of the `ErrorHandler` class prints error messages in the browser console window.

To create a custom error handler for our application, we need to sub-class the `ErrorHandler` class and provide our tailored implementation for error logging:

1. Create a file named `app-error-handler.ts` in the `src\app` folder of the Angular CLI workspace.
2. Open the file and add the following import statements:

```
import { HttpErrorResponse, HttpStatusCode } from '@angular/common/
http';
import { ErrorHandler, Injectable } from '@angular/core';
```

3. Create a TypeScript class that implements the `ErrorHandler` interface:

```
@Injectable()  
export class AppErrorHandler implements ErrorHandler {}
```

The `AppErrorHandler` class must be decorated with the `@Injectable()` decorator because we will provide it later in the application configuration file.

4. Implement the `handleError` method from the `ErrorHandler` interface as follows:

```
handleError(error: any): void {  
    const err = error.rejection || error;  
    let message = '';  
  
    if (err instanceof HttpErrorResponse) {  
        switch(err.status) {  
            case 0:  
                message = 'Client error';  
                break;  
            case HttpStatusCode.InternalServerError:  
                message = 'Server error';  
                break;  
            case HttpStatusCode.BadRequest:  
                message = 'Request error';  
                break;  
            default:  
                message = 'Unknown error';  
        }  
    } else {  
        message = 'Application error';  
    }  
  
    console.error(message, err);  
}
```

In the preceding method, we check if the `error` object contains a `rejection` property. Errors originating from the `Zone.js` library, which is responsible for the change detection in Angular, encapsulate the actual error inside that property.

After extracting the error in the `err` variable, we check to see if it is an HTTP error using the `HttpErrorResponse` type. This check will eventually catch any errors from HTTP calls using the `throwError` RxJS operator. All other errors are treated as application errors that occur on the client side.

5. Open the `app.config.ts` file and import the `ErrorHandler` class from the `@angular/core` npm package:

```
import { ApplicationConfig, ErrorHandler, provideZoneChangeDetection
} from '@angular/core';
```

6. Import the custom error handler we created in the `app-error-handler.ts` file:

```
import { AppErrorHandler } from './app-error-handler';
```

7. Register the `AppErrorHandler` class as the global error handler of the application by adding it to the `providers` array of the `appConfig` variable:

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
    { provide: APP_SETTINGS, useValue: appSettings },
    { provide: ErrorHandler, useClass: AppErrorHandler }
  ]
};
```

To investigate the behavior of the global application error handler, execute the following steps:

1. Run the `ng serve` command to start the application.
2. Disconnect your computer from the internet.
3. Navigate to `http://localhost:4200`.
4. Open the browser developer tools and inspect the output of the console window:

```

✖ ► Client error
  ▼ HttpErrorResponse {headers: _HttpHeaders, status: 0, statusText: 'Unknown Error', url:
  'https://fakestoreapi.com/products', ok: false, ...} i
    ► error: ProgressEvent {isTrusted: true, lengthComputable: false, loaded: 0, total: 0, t
    ► headers: _HttpHeaders {normalizedNames: Map(0), lazyUpdate: null, headers: Map(0)}
      message: "Http failure response for https://fakestoreapi.com/products: 0 Unknown Error
      name: "HttpErrorResponse"
      ok: false
      status: 0
      statusText: "Unknown Error"
      url: "https://fakestoreapi.com/products"
    ► [[Prototype]]: HttpResponseBase

```

Figure 11.1: Application error

One of the most common HTTP errors in a web enterprise application is the **401 Unauthorized** response error. We will learn how to handle this specific error in the following section.

Responding to the 401 Unauthorized error

The 401 Unauthorized error in an Angular application can occur in the following cases:

- The user does not provide the correct credentials while logging in to the application
- The authentication token provided when the user logged in to the application has expired

A good place to handle the 401 Unauthorized error is inside an HTTP interceptor responsible for authentication. In *Chapter 8, Communicating with Data Services over HTTP*, we learned how to create an authentication interceptor for passing the authorization token to every HTTP request. To handle the 401 Unauthorized error, the `auth.interceptor.ts` file could be modified as follows:

```

import { HttpErrorResponse, HttpInterceptorFn, HttpStatusCode } from '@
angular/common/http';
import { inject } from '@angular/core';
import { AuthService } from './auth.service';
import { catchError, EMPTY, throwError } from 'rxjs';

export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const authService = inject(AuthService);
  const authReq = req.clone({
    setHeaders: { Authorization: 'myToken' }
  });
  return next(authReq).pipe(
    catchError((error: HttpErrorResponse) => {

```

```
    if (error.status === HttpStatusCode.Unauthorized) {
        authService.logout();
        return EMPTY;
    } else {
        return throwError(() => error);
    }
}
);
};
```

The interceptor will call the `logout` method of the `AuthService` class when a 401 Unauthorized error occurs and return an `EMPTY` observable to stop emitting data. It will use the `throwError` operator to bubble the error to the global error handler in all other errors. As we have already seen, the global error handler will examine the returned error and take action according to the status code.

As we saw in the global error handler we created in the previous section, some errors are unrelated to the interaction with the HTTP client. There are application errors that occur on the client side, and we will learn how to understand them in the following section.

Demystifying framework errors

Application errors that originate on the client side in an Angular application can have many causes. One of them is the interaction of our source code with the Angular framework. Developers like to try new things and approaches while building applications. Sometimes, things will go well but, other times, they may cause errors in an application.

The Angular framework provides a mechanism for reporting some of these common errors with the following format:

NGWXYZ: {Error message}.<Link>

Let's analyze the preceding error format:

- NG: Indicates that it is an Angular error to differentiate between other errors originating from TypeScript and the browser
- W: A single-digit number that indicates the type of the error. 0 represents a runtime error, and all other numbers from 1 to 9 represent a compiler error
- X: A single-digit number that indicates the category of the framework runtime area, such as change detection, dependency injection, and template
- YZ: A two-digit code used to index the specific error

- `{Error message}`: The actual error message
- `<Link>`: A link to the Angular documentation that provides more information about the specified error

Error messages that conform to the preceding format are displayed in the browser console as they happen. Let's see an error example using the `ExpressionChangedAfterChecked` error, the most famous error in Angular applications:

1. Open the `app.component.ts` file and import the `AfterViewInit` artifact from the `@angular/core` npm package:

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

2. Add the `AfterViewInit` in the list of implemented interfaces:

```
export class AppComponent implements AfterViewInit
```

3. Create the following `title` property in the `AppComponent` class:

```
title = '';
```

4. Implement the `ngAfterViewInit` method and change the `title` property inside the method body:

```
ngAfterViewInit(): void {
  this.title = this.settings.title;
}
```

5. Open the `app.component.html` file and bind the `title` property to the `<h2>` HTML element:

```
<h2>{{ title }}</h2>
```

6. Run the `ng serve` command and navigate to `http://localhost:4200`.

Initially, everything looks to work correctly. The value of the `title` property is displayed on the page correctly.

7. Open the browser developer tools and inspect the console window:

```
Application error RuntimeError: NG0100:
ExpressionChangedAfterItHasBeenCheckedError: Expression has changed
after it was checked. Previous value: ''. Current value: 'My
e-shop'. Expression location: _AppComponent component. Find more at
https://angular.dev/errors/NG0100
```

The preceding message indicates that changing the value of the `title` property caused the error.

8. Clicking on the <https://angular.dev/errors/NG0100> link will redirect us to the appropriate error guide in the Angular documentation for more information. The error guide explains the specific error and describes how to fix the problem in our application code.

When we understand the error messages that originate from the Angular framework, we can fix them easily.

Summary

Handling errors during runtime or development is crucial for every Angular application. In this chapter, we learned how to handle errors that occur during the runtime of an Angular application, such as HTTP or client-side errors. We also learned how to understand and fix application errors thrown by the Angular framework.

In the next chapter, we will learn how to skin our application to look more beautiful with the help of Angular Material. Angular Material has many components and styles that are ready for you to use in your projects. So, let's give your Angular project the love it deserves.

12

Introduction to Angular Material

When developing a web application, you must decide how to create your **user interface (UI)**. It should ideally use proper contrasting colors, have a consistent look and feel, be responsive, and work well on different devices and browsers. In short, there are many things to consider regarding UI and UX. Many developers consider creating the UI/UX a daunting task and turn to UI frameworks that do much of the heavy lifting. Some frameworks are used more than others, namely **Bootstrap** and **Tailwind CSS**. However, **Angular Material**, a framework based on Google's **Material Design** techniques, has gained popularity. In this chapter, we will explain what Material Design is and how Angular Material uses it to provide a component UI library for the Angular framework. We will also learn to use various Angular Material components by applying them in our e-shop application.

In this chapter, we will be doing the following:

- Introducing Material Design
- Introducing Angular Material
- Integrating UI components

Technical requirements

The chapter contains various code samples to walk you through the concept of Angular Material. You can find the related source code in the ch12 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Introducing Material Design

Material Design is a design language developed by Google with the following goals in mind:

- Develop a single underlying system, allowing a unified experience across platforms and device sizes.
- Mobile precepts are fundamental, but touch, voice, mouse, and keyboard are all first-class input methods.

The purpose of a design language is to have the user deal with how the UI and user interaction should look and feel across devices. Material Design is based on three main principles:

- **Material is the metaphor:** It is inspired by the physical world with different textures and mediums, such as paper and ink.
- **Bold, graphic, and intentional:** It is guided by different print design methods, such as typography, grids, and color, to create an immersive experience for the user.
- **Motion provides meaning:** Elements are displayed on the screen by creating animations and interactions that reorganize the environment.

Material Design has much theory behind it, and proper documentation on the topic is available should you wish to delve further. You can find more information at the official documentation site: <https://material.io>

A design language alone isn't that interesting if you are not a designer. In the following section, we will learn how Angular developers can benefit from Material Design using the Angular Material library.

Introducing Angular Material

The Angular Material library was developed to implement Material Design for the Angular framework. It is based on the following concepts:

- **Sprint from zero to app:** The intention is to make it easy for you, as an application developer, to hit the ground running. The effort needed to set it up should be minimal.
- **Fast and consistent:** Performance has been a significant focus point, and Angular Material is guaranteed to work well on all major browsers.
- **Versatile:** Many themes should be easily customizable, and there is also great support for localization and internationalization.

- **Optimized for Angular:** The fact that the Angular team has built it means that support for Angular is a big priority.

The library is split into the following main parts:

- **Components:** Many UI components, such as different kinds of input, buttons, layout, navigation, modals, and other ways to show tabular data, are in place to help you be successful.
- **Themes:** The library comes with preinstalled themes, but there is also a theming guide if you want to create your own at <https://material.angular.io/guide/theming>.



Every part and component of the Angular Material library encapsulates web accessibility best techniques out of the box.

The core of the Angular Material library is the **Angular CDK**, which is a collection of tools that implement similar interaction patterns unrelated to any presentation style. The behavior of Angular Material components has been designed using the Angular CDK. The Angular CDK is so abstract that you can use it to create custom components. You should seriously consider it if you are a UI library author.

We have covered all the basic theory about Angular Material, so let's put it into practice in the following section by integrating it with an Angular application.

Installing Angular Material

The Angular Material library is an npm package. To install it, we need to manually execute the `npm install` command and import several Angular artifacts into our Angular application. The Angular team has automated these interactions by creating the necessary schematics to install it using the Angular CLI.



You will need the source code of the Angular application we created in *Chapter 11, Handling Application Errors*, to follow along with the rest of the chapter.

We can use the `ng add` command of the Angular CLI to install Angular Material into our e-shop application:

1. Run the following command in the current Angular CLI workspace:

```
ng add @angular/material
```

The Angular CLI will find the latest stable version of the Angular Material library and prompt us to download it.



In this book, we work with Angular Material 19, which is compatible with Angular 19. If the version that prompts you is different, you should run the command `ng add @angular/material@19` to install the latest Angular Material 19 to your system.

2. After the download is complete, it will ask us whether we want to use a prebuilt theme for our Angular application or a custom one:

```
Choose a prebuilt theme name, or "custom" for a custom theme: (Use arrow keys)
```

Accept the default value, **Azure/Blue**, by pressing *Enter*.

3. After selecting a theme, the Angular CLI will ask if we want to set up global typography styles in our application. Typography refers to how the text is arranged in our application:

```
Set up global Angular Material typography styles? (y/N)
```

We want to keep our application as simple as possible, so accept the default value, **No**, by pressing *Enter*.



Angular Material typography is based on Material Design guidelines and uses the Roboto Google Font for styling.

4. The next question is about animations. Animation isn't strictly required, but we want our application to display a beautiful animation when we click a button or open a modal dialog:

```
Include the Angular animations module? (Use arrow keys)
```

Accept the default value, **Include and enable animations**, by pressing *Enter*.

The Angular CLI will start installing and configuring Angular Material into our application. It will scaffold and import all necessary artifacts so we can start working with Angular Material immediately:

- `angular.json`: It adds the theme stylesheet file in the configuration file of the Angular CLI workspace:

```
"styles": [
  "@angular/material/prebuilt-themes/azure-blue.css",
  "src/styles.css"
]
```

- `package.json`: It adds the `@angular/cdk` and `@angular/material` npm packages.
- `index.html`: It adds the stylesheet files of the Roboto fonts and the Material icons in the main HTML file.
- `styles.css`: It adds the necessary global CSS styles for the `<html>` and the `<body>` tags:

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif;
}
```

- `app.config.ts`: It enables animations in the application configuration file:

```
import { provideHttpClient } from '@angular/common/http';
import { ApplicationConfig, ErrorHandler, provideZoneChangeDetection
} from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { APP_SETTINGS, appSettings } from './app.settings';
import { AppErrorHandler } from './app-error-handler';
import { provideAnimationsAsync } from '@angular/platform-browser/
animations/async';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    provideHttpClient(),
    { provide: APP_SETTINGS, useValue: appSettings },
  ]
}
```

```
{ provide: ErrorHandler, useClass: AppErrorHandler },
provideAnimationsAsync()
]
};
```

After the process is finished, we can begin adding UI components from the Angular Material library into our application.

Adding UI components

The button component is one of the most used components from the Angular Material library. As an example, we will learn how easy it is to add a button component to our e-shop application. Before we can use it in our Angular application, we must remove all CSS styles for the native `<button>` tag that we have used so far:

1. Open the `styles.css` file and remove the `button`, `button:hover`, and `button:disabled` CSS styles.
2. Open the `product-detail.component.css` file and remove the `--button-accent` variable from the `button.secondary` and `button.delete` styles.
3. Remove the `.button-group` CSS style completely.
4. Add a color in the `button.delete` style:

```
button.delete {
  display: inline;
  margin-left: 5px;
  color: brown;
}
```

To start using a UI component from the Angular Material library, we must import its corresponding Angular component. Let's see how this is done by adding a button component in the authentication component of the Angular application:

1. Open the `auth.component.ts` file and add the following `import` statement to use Angular Material buttons:

```
import { MatButton } from '@angular/material/button';
```

We do not import directly from the `@angular/material` package because every component has a dedicated namespace. The button component can be found in the `@angular/material/button` namespace.



Angular Material components can also be used by importing their respective module, such as `MatButtonModule` for buttons. It is recommended to import the components directly though, because it helps us stay consistent with modern Angular patterns. However, we will see that some features require too many components to import. In those cases, it is acceptable to import the module directly.

2. Add the `MatButton` class in the `imports` array of the `@Component` decorator:

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

3. Open the `auth.component.html` file and add the `mat-button` directive in the `<button>` HTML elements:

```
@if (!authService.isLoggedIn()) {
  <button mat-button (click)="login()">Login</button>
} @else {
  <button mat-button (click)="logout()">Logout</button>
}
```

In the preceding template, the `mat-button` directive, in essence, modifies the `<button>` element so that it appears and behaves as a Material Design button.

If we run the `ng serve` command and navigate to `http://localhost:4200`, we will notice that the button style is different than before. It looks more like a link, which is the default appearance of a Material button. In the following section, we will learn about theming and variations of the button component.

Theming UI components

The Angular Material library comes with four built-in themes:

- Azure/Blue
- Rose/Red

- Magenta/Violet
- Cyan/Orange

When we add Angular Material to an Angular application, we can choose which of the preceding themes we want to apply. We can always change it by modifying the included CSS stylesheet file in the `angular.json` configuration file. Here's an example:

```
"styles": [  
  "/@angular/material/prebuilt-themes/azure-blue.css",  
  "src/styles.css"  
]
```

As we saw in the preceding section, the button component is displayed as a link. The `mat-button` directive displays a background color only when we hover over the button. To set the background color permanently, we must use the `mat-flat-button` directive as follows:

```
@if (!authService.isLoggedIn()) {  
  <button mat-flat-button (click)="login()">  
    Login  
  </button>  
} @else {  
  <button mat-flat-button (click)="logout()">  
    Logout  
  </button>  
}
```

Now that we know how to interact with the button component in an Angular application let's learn some of its variations:

1. Open the `product-create.component.ts` file and add the following import statement:

```
import { MatButton } from '@angular/material/button';
```

2. Add the `MatButton` class in the `imports` array of the `@Component` decorator:

```
@Component({  
  selector: 'app-product-create',  
  imports: [ReactiveFormsModule, MatButton],  
  templateUrl: './product-create.component.html',  
  styleUrls: ['./product-create.component.css'  
})
```

3. Open the `product-create.component.html` file and add the `mat-raised-button` directive in the `<button>` HTML element:

```
<button  
  mat-raised-button  
  type="submit"  
  [disabled]="productForm.invalid">  
  Create  
</button>
```

The `mat-raised-button` directive will add a shadow to the button element:

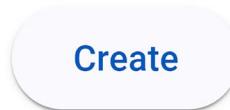


Figure 12.1: Raised button

4. Open the `product-detail.component.ts` file and repeat steps 1 and 2.
5. Open the `product-detail.component.html` file and add the `mat-stroked-button` directive in the Change button:

```
<button  
  mat-stroked-button  
  class="secondary"  
  type="submit"  
  [disabled]="priceForm.invalid">  
  Change  
</button>
```

The `mat-stroked-button` directive adds a border around the button element:



Figure 12.2: Stroked button

6. Remove the `<div>` HTML element with the `button-group` class and add the `mat-raised-button` directive in both `<button>` HTML elements:

```
@if (authService.isLoggedIn()) {
```

```

<button
  mat-raised-button
  (click)="addToCart(product.id)">
  Add to cart
</button>
}
<button
  mat-raised-button
  class="delete"
  (click)="remove(product)">
  Delete
</button>

```

The two buttons appear as follows when we run the application:



Figure 12.3: Product detail action buttons

7. Open the `product-list.component.ts` file and add the following import statements:

```

import { MatMiniFabButton } from '@angular/material/button';
import { MatIcon } from '@angular/material/icon';

```

8. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```

@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    AsyncPipe,
    RouterLink,
    MatMiniFabButton,
    MatIcon
  ],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})

```

9. Open the `product-list.component.html` file and replace the anchor element that navigates to the product create component with the following HTML snippet:

```
<button mat-mini-fab routerLink="new">  
  <mat-icon>add</mat-icon>  
</button>
```

The `mat-mini-fab` directive displays a square button with rounded corners and an icon indicated by the `<mat-icon>` HTML element. The text of the `<mat-icon>` element corresponds to the add icon name from the Material Design icons collection:



Figure 12.4: FAB button

Theming in Angular Material is so extensive that we can use existing CSS variables to create custom themes, a topic that is out of the scope of this book.

To continue our journey through the land of styling with Angular Material, we will learn how to integrate various UI components in the next section.

Integrating UI components

Angular Material contains a lot of UI components organized in categories at <https://material.angular.io/components/categories>. In this chapter, we will explore a subset of the preceding collection that can be grouped into the following categories:

- **Form controls:** These can be used inside an Angular form, such as autocomplete, input, and drop-down list.
- **Navigation:** These provide navigation capabilities, such as a header and footer.
- **Layout:** These define how data is represented, such as a card or table.
- **Popups and overlays:** These are overlay windows that display information and can block any user interaction until they are dismissed in any way.

In the following sections, we will explore each category in more detail.

Form controls

We learned in *Chapter 10, Collecting User Data with Forms*, that form controls are about collecting input data in different ways and taking further action, such as sending data to a backend API over HTTP.

There are quite a few form controls in the Angular Material library of varying types, namely the following:

- **Autocomplete:** Enables the user to start typing in an input field and be presented with suggestions while typing. It helps to narrow down the possible values that the input can take.
- **Checkbox:** A classic checkbox representing a state either checked or unchecked.
- **Date picker:** Allows the user to select a date in a calendar.
- **Input:** A classic input control enhanced with meaningful animation while typing.
- **Radio button:** A classic radio button enhanced with animations and transitions while editing to create a better user experience.
- **Select:** A drop-down control that prompts the user to select one or more items from a list.
- **Slider:** Enables the user to increase or decrease a value by pulling a slider button to either the right or the left.
- **Slide toggle:** A switch the user can slide to set on or off.
- **Chips:** A list that displays, selects, and filters items.

In the following sections, we will examine some of these form controls in more detail. Let's begin with the input component.

Input

The input component is usually attached to an `<input>` HTML element. We can also add the ability to display errors in the input field.

Before we can use the input component in our Angular application, we must remove all CSS styles for the native `<input>` tag that we have used so far:

1. Open the `styles.css` file and remove any CSS styles referencing the `input` tag.
2. Remove the `input` CSS style from the `product-create.component.css` and `cart.component.css` files.

To learn how to use the input component, we will integrate it into our application components:

1. Open the `product-create.component.ts` file and add the following import statements:

```
import { MatInput } from '@angular/material/input';
import { MatFormField, MatError, MatLabel } from '@angular/material/form-field';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-create',
  imports: [
    ReactiveFormsModule,
    MatButtonModule,
    MatInput,
    MatFormField,
    MatError,
    MatLabel
  ],
  templateUrl: './product-create.component.html',
  styleUrls: ['./product-create.component.css']
})
```

3. Open the `product-create.component.html` file and replace the `<div>` tags of the `<input>` HTML elements as follows:

```
<mat-form-field>
  <mat-label>Title</mat-label>
  <input formControlName="title" matInput required />
  <mat-error>Title is required</mat-error>
</mat-form-field>
<mat-form-field>
  <mat-label>Price</mat-label>
  <input formControlName="price" matInput type="number" required />
  @if (productForm.controls.price.touched && productForm.controls.
  price.hasError('required')) {
    <mat-error>Price is required</mat-error>
  }
}
```

```

@if (productForm.controls.price.touched && productForm.controls.
price.hasError('min')) {
    <mat-error>Price should be greater than 0</mat-error>
}
@if (productForm.controls.price.touched && productForm.controls.
price.hasError('priceMaximum')) {
    <mat-error>Price must be smaller or equal to 1000</mat-error>
}
</mat-form-field>
```

In the preceding HTML snippet, we use the `matInput` directive to indicate that an `<input>` HTML element is an Angular Material input component. A form control in Angular Material must be enclosed in a `<mat-form-field>` element.

We have replaced all `<label>` HTML elements with `<mat-label>` elements. A `<mat-label>` HTML element is a label that targets a specific Angular Material form control.

The `<mat-error>` element displays error messages in form controls when Angular triggers validation errors. It is shown by default when the status of the form control is invalid. In all other cases, we can use an `@if` block to control when the `<mat-error>` element will be displayed.

4. Open the `global.styles.css` file and add the following CSS style:

```

mat-form-field {
    width: 100%;
}
```

In the preceding snippet, we configure `mat-form-field` elements to take all the available width.

5. Run the `ng serve` command to start the application and navigate to `http://localhost:4200/products/new`. Focus on the appearance of the input fields:

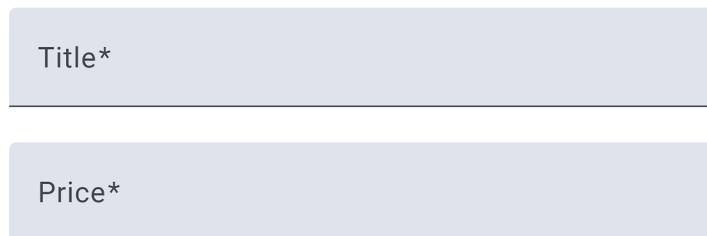


Figure 12.5: Input component

In the preceding figure, the label of each form control is suffixed by an asterisk. The asterisk is a common indication that the form control must have a value. Angular Material automatically adds it by recognizing the required attribute on the `<input>` HTML element.

6. Open the `cart.component.ts` file and repeat steps 1 and 2, but do not include the `MatError` class.
7. Open the `cart.component.html` file and modify the contents of the `@for` block as follows:

```
@for(product of cartForm.controls.products.controls; track $index) {  
  <mat-form-field>  
    <mat-label>{{products[$index].title}}</mat-label>  
    <input  
      [formControlName]="$index"  
      placeholder="{{products[$index].title}}"  
      type="number"  
      matInput />  
  </mat-form-field>  
}
```

The remaining component of our application that contains an `<input>` HTML element is the product detail component. The product detail component is a special case of an Angular Material input because we must group it with the button that changes the product price:

1. Open the `product-detail.component.ts` file and modify the `import` statement from the Angular Material npm package as follows:

```
import { MatButton, MatIconButton } from '@angular/material/button';  
import { MatInput } from '@angular/material/input';  
import { MatFormField, MatError, MatSuffix } from '@angular/  
material/form-field';  
import { MatIcon } from '@angular/material/icon';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```
@Component({  
  selector: 'app-product-detail',  
  imports: [  
    CommonModule,  
    FormsModule,  
    PriceMaximumDirective,  
    MatButton,
```

```
    MatInput,
    MatFormField,
    MatError,
    MatIcon,
    MatSuffix,
    MatIconButton
  ],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css'
})
```

3. Open the product-detail.component.html file and modify the <form> HTML element as follows:

```
<form (ngSubmit)="changePrice(product)" #priceForm="ngForm">
  <mat-form-field>
    <input
      placeholder="New price"
      type="number"
      name="price"
      required min="1"
      appPriceMaximum threshold="500"
      matInput
      #priceCtrl="ngModel"
      [(ngModel)]="price" />
  <button
    mat-icon-button
    matSuffix
    type="submit"
    [disabled]="priceForm.invalid">
    <mat-icon>edit</mat-icon>
  </button>
  @if (priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.
  hasError('min'))) {
    <mat-error>Please enter a valid price</mat-error>
  }
  @if (priceCtrl.dirty && priceCtrl.hasError('priceMaximum')) {
```

```
<mat-error>Price must be smaller or equal to 500</mat-error>
}
</mat-form-field>
</form>
```

In the preceding snippet, we modified the button that changes the price so that it displays a pencil icon, and it is placed in line with the `<input>` HTML element.

The `mat-icon-button` directive indicates that the button will not have any text. Instead, it will display an icon defined by the `<mat-icon>` HTML element. The `matSuffix` directive positions the button inline and at the end of the `<input>` HTML element.

4. Navigate to the product list in the browser and select one product. The input for changing the product price should be the following:

A screenshot of a user interface showing an input field with the placeholder "New price". To the right of the input field is a small grey rectangular button containing a white pencil icon, which serves as an inline edit button.

New price

Figure 12.6: Input component with inline button

In the following section, we will learn how to use an Angular Material select component to choose a category in the product create component.

Select

The select component works similarly to the native `<select>` HTML element. It displays a dropdown element with a list of options for users.

We will add one in the product create component to select the category of a new product:

1. Open the `product-create.component.ts` file and add the following import statement:

```
import { MatSelect, MatOption } from '@angular/material/select';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-create',
  imports: [
    ReactiveFormsModule,
    MatButton,
    MatInput,
```

```

        MatFormField,
        MatError,
        MatLabel,
        MatSelect,
        MatOption
    ],
    templateUrl: './product-create.component.html',
    styleUrls: ['./product-create.component.css']
})

```

3. Open the `product-create.component.html` file and replace the `<div>` HTML element that encloses the `<select>` element with the following HTML snippet:

```

<mat-form-field>
  <mat-label>Category</mat-label>
  <mat-select formControlName="category">
    <mat-option value="electronics">Electronics</mat-option>
    <mat-option value="jewelery">Jewelery</mat-option>
    <mat-option>Other</mat-option>
  </mat-select>
</mat-form-field>

```

In the preceding snippet, we replaced the `<select>` and the `<option>` HTML elements with the `<mat-select>` and the `<mat-option>` elements, respectively.

4. Navigate to `http://localhost:4200/products/new` and click on the **Category** drop-down list:

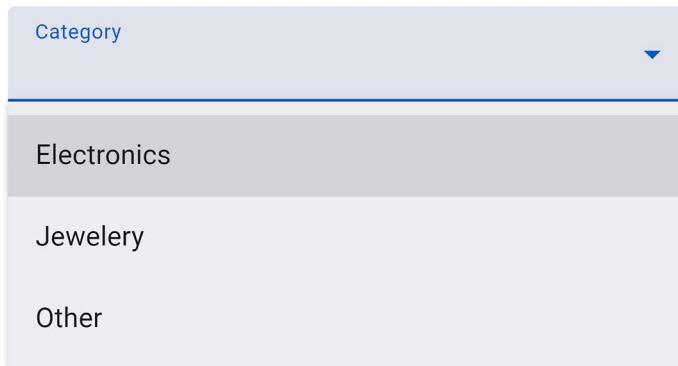


Figure 12.7: Select component

The product details component displays the product category as a paragraph element with a specific CSS class. In the following section, we will learn how to represent the product category with the Angular Material chips component.

Chips

The chips component is often used to display information grouped by a specific property. It can also provide data filtering and selection capabilities. We can use chips in our application to display the category in the product details component.



Our products only have one category, but chips would make more sense if we had additional categories assigned to our products.

Let's get started:

1. Open the `product-detail.component.ts` file and add the following `import` statement:

```
import { MatChipSet, MatChip } from '@angular/material/chips';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-detail',
  imports: [
    CommonModule,
    FormsModule,
    PriceMaximumDirective,
    MatButton,
    MatInput,
    MatFormField,
    MatError,
    MatIcon,
    MatSuffix,
    MatIconButton,
    MatChipSet,
    MatChip
  ],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css']
})
```

3. Open the `product-detail.component.html` file and replace the `<div>` HTML element that contains the `pill-group` class with the following content:

```
<mat-chip-set>
  <mat-chip>{{ product.category }}</mat-chip>
</mat-chip-set>
```

The `<mat-chip>` HTML element indicates a chip component. Chips must always be enclosed using a container element. The simplest form of a chips container is the `<mat-chip-set>` element.

4. Open the `product-detail.component.css` file and add the following CSS style:

```
mat-chip-set {
  margin-bottom: 1.375rem;
}
```

5. Run the `ng serve` command to start the application and select a product from the list. The category should, for example, look like the following:



electronics

Figure 12.8: Chips component

The chips component completes our exploration of the Angular Material form controls. In the following section, we will get hands-on experience by styling the navigation layout of the application.

Navigation

There are different ways of navigating in an Angular application, such as clicking a link or a menu item. Angular Material offers the following components for this type of interaction:

- **Menu:** A pop-up list where you can choose from a predefined set of options.
- **Sidenav:** A component that acts as a menu docked to the left or the right of the page. It can be presented as an overlay over the application while dimming the application content.
- **Toolbar:** A standard toolbar that allows the user to reach commonly used actions.

In this section, we will demonstrate how to use the toolbar component. We will convert the `<header>` and `<footer>` HTML elements of the main application component to Angular Material toolbars.

To create a toolbar, we will go through the following steps:

1. Open the `app.component.ts` file and add the following import statements:

```
import { MatToolbarRow, MatToolbar } from '@angular/material/toolbar';
import { MatButton } from '@angular/material/button';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator and remove the `RouterLinkActive` class:

```
@Component({
  selector: 'app-root',
  imports: [
    RouterOutlet,
    RouterLink,
    CopyrightDirective,
    AuthComponent,
    MatToolbarRow,
    MatToolbar,
    MatButton
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

3. Open the `app.component.html` file and modify the `<header>` HTML element as follows:

```
<header>
  <mat-toolbar>
    <mat-toolbar-row>
      <h2>{{ settings.title }}</h2>
      <span class="spacer"></span>
      <button mat-button routerLink="/products">Products</button>
      <button mat-button routerLink="/cart">My Cart</button>
      <button mat-button routerLink="/user">My Profile</button>
      <app-auth></app-auth>
    </mat-toolbar-row>
  </mat-toolbar>
</header>
```

In the preceding template, we add the main application links and the authentication component inside a `<mat-toolbar>` element. The toolbar component consists of a single row indicated by the `<mat-toolbar-row>` HTML element.

4. Open the `app.component.css` file and remove the CSS style for the header tag and the `menu-links`.
5. If we run the application using the `ng serve` command, we will see the new toolbar of our application at the top of the page:

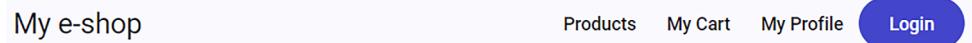


Figure 12.9: Application header

6. Now, modify the `<footer>` HTML element to convert it to an Angular Material toolbar component:

```
<footer>
  <mat-toolbar>
    <mat-toolbar-row>
      <span appCopyright> - v{{ settings.version }}</span>
    </mat-toolbar-row>
  </mat-toolbar>
</footer>
```

7. Save changes, wait for the application to refresh, and observe the toolbar at the bottom of the application:

Copyright ©2024 All Rights Reserved - v1.0

Figure 12.10: Application footer

The toolbar component is fully customizable, and we can adjust it according to the application's needs. We can add icons and even create toolbars with content in multiple rows. Now that you know the basics of creating a simple toolbar, you can explore further possibilities.

In the following section, we will learn how to lay out content differently inside our application.

Layout

When we refer to the layout, we discuss how we place content in our templates. Angular Material gives us different components for this purpose:

- **List:** Visualizes the content as a list of items. It can be enriched with links and icons and even multiline.
- **Grid list:** Helps us arrange the content in blocks. We only need to define the number of columns; the component will fill the visual space.
- **Card:** Wraps content and adds a box shadow. We can define a header for it as well.
- **Tabs:** Divides up the content into different tabs.
- **Stepper:** Divides up the content into wizard-like steps.
- **Expansion panel:** Enables us to place the content in a list-like way with a title for each item. Items can only be expanded one at a time.
- **Table:** Represents data in a tabular format with rows and columns.

In this book, we will cover the card and table components.

Card

We will learn how to display each product in the list as a card:

1. Open the `product.ts` file and add an `image` property to the `Product` interface:

```
export interface Product {  
    id: number;  
    title: string;  
    price: number;  
    category: string;  
    image: string;  
}
```

The `image` property is a URL that points to the product image file in the Fake Store API.

2. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { MatCardModule } from '@angular/material/card';
```

3. Add the `MatCardModule` class in the `imports` array of the `@Component` decorator:

```
@Component({  
    selector: 'app-product-list',  
    imports: [  
        SortPipe,  
        AsyncPipe,  
        RouterLink,
```

```
        MatMiniFabButton,
        MatIcon,
        MatCardModule
    ],
    templateUrl: './product-list.component.html',
    styleUrls: ['./product-list.component.css'
})
```



The Angular Material card component consists of many other components and directives. We choose to import the whole Angular module because it would not be convenient to import them all individually.

4. Open the `product-list.component.html` file and replace the unordered list element with the following HTML snippet:

```
@for (product of products | sort; track product.id) {
  <mat-card [routerLink]=[product.id]>
    <mat-card-header>
      <mat-card-title-group>
        <mat-card-title>{{ product.title }}</mat-card-title>
        <mat-card-subtitle>{{ product.category }}</mat-card-
subtitle>
        <img mat-card-sm-image [src]=“product.image” />
      </mat-card-title-group>
    </mat-card-header>
  </mat-card>
} @empty {
  <p>No products found!</p>
}
```

An Angular Material card component consists of a header, indicated by the `<mat-card-header>` HTML element. The header component contains a `<mat-card-title-group>` HTML element that aligns the card title, subtitle, and image into a single section. The card title indicated by the `<mat-card-title>` HTML element displays the product title. The card subtitle indicated by the `<mat-card-subtitle>` HTML element displays the product category. Finally, the product image is displayed by attaching the `mat-card-sm-image` directive to an `` HTML element. The `sm` keyword in the directive indicates that we want to render a small size of the image.



Angular Material also supports `md` and `lg` for medium and large sizes, respectively.

5. Open the `product-list.component.css` file and add the following CSS style:

```
mat-card {  
  margin: 1.375rem;  
  cursor: pointer;  
}
```

6. Run the application using the `ng serve` command and navigate to `http://localhost:4200`:

Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
men's clothing



John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
jewelry



Mens Casual Premium Slim Fit T-Shirts
men's clothing



Mens Casual Slim Fit
men's clothing



Mens Cotton Jacket
men's clothing



Figure 12.11: Product list card representation

You can explore more options for the card component by navigating to <https://material.angular.io/components/card/overview>.

In the following section, we will learn how to switch the product list to a tabular view.

Data table

The table component from the Angular Material library enables us to display our data in columns and rows. To create a table, we must import the `MatTableModule` class from the `@angular/material/table` namespace.



The Angular Material data table consists of many other components and directives. We choose to import the whole Angular module because it would not be convenient to import them all individually.

Let's get started:

1. Open the `product-list.component.ts` file and import the `CurrencyPipe` and the `MatTableModule` artifacts:

```
import { AsyncPipe, CurrencyPipe } from '@angular/common';
import { MatTableModule } from '@angular/material/table';
```

2. Add the preceding imported classes to the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    AsyncPipe,
    CurrencyPipe,
    RouterLink,
    MatMiniFabButton,
    MatIcon,
    MatCardModule,
    MatTableModule
  ],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

3. Create the following property in the `ProductListComponent` class to define the table column names:

```
columnNames = ['title', 'price'];
```

The name of each column matches a property from the `Product` interface.

4. Open the `product-list.component.html` file and add the following snippet after the `@for` block:

```
<table mat-table [dataSource]="products"></table>
```

An Angular Material table is a standard `<table>` HTML element with the `mat-table` directive attached.

The `dataSource` property of the `mat-table` directive defines the data we want to display on the table. It can be any data that can be enumerated, such as an array. In our case, we bind it to the `products` template reference variable.

5. Add an `<ng-container>` element for each column we want to display:

```
<table mat-table [dataSource]="products">
  <ng-container matColumnDef="title">
    <th mat-header-cell *matHeaderCellDef>Title</th>
    <td mat-cell *matCellDef="let product">
      <a [routerLink]=[product.id]>{{ product.title }}</a>
    </td>
  </ng-container>
  <ng-container matColumnDef="price">
    <th mat-header-cell *matHeaderCellDef>Price</th>
    <td mat-cell *matCellDef="let product">{{ product.price | currency }}</td>
  </ng-container>
</table>
```



The `<ng-container>` element is a unique-purpose element that groups elements with similar functionality. It does not interfere with the styling of the child elements, nor is it rendered on the screen.

The `<ng-container>` element uses the `matColumnDef` directive to set the name of the specific column.



The value of the `matColumnDef` directive must match with a value from the `columnNames` component property; otherwise, the application will throw an error that it cannot find the name of the defined column.

It contains a `<th>` HTML element with a `mat-header-cell` directive that indicates the header of the cell and a `<td>` HTML element with a `mat-cell` directive for the data of the cell. The `<td>` HTML element uses the `matCellDef` directive to create a local template variable for the current row data that we can use later.

6. Add the following snippet after the `<ng-container>` elements:

```
<tr mat-header-row *matHeaderRowDef="columnNames"></tr>
<tr mat-row *matRowDef="let row; columns: columnNames;"></tr>
```

In the preceding snippet, we define the header row of the table that displays column names and the actual rows that contain data.

If we run the application, the output should be the following:

Title	Price
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops	\$109.95
John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet	\$695.00
Mens Casual Premium Slim Fit T-Shirts	\$22.30
Mens Casual Slim Fit	\$15.99
Mens Cotton Jacket	\$55.99
Pierced Owl Rose Gold Plated Stainless Steel Double	\$10.99
SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s	\$109.00
Solid Gold Petite Micropave	\$168.00
WD 2TB Elements Portable External Hard Drive - USB 3.0	\$64.00
White Gold Plated Princess	\$9.99

Figure 12.12: Table component

The product list component displays the card and the table representation of data simultaneously. We will use the button toggle component from Angular Material to distinguish between them. The button toggle component toggles buttons on or off according to a specific condition:

1. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { MatButtonToggle, MatButtonToggleGroup } from '@angular/material/button-toggle';
```

2. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    AsyncPipe,
    CurrencyPipe,
    RouterLink,
    MatMiniFabButton,
    MatIcon,
    MatCardModule,
    MatTableModule,
    MatButtonToggle,
    MatButtonToggleGroup
  ],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

3. Open the `product-list.component.html` file and add the following HTML snippet inside the `<div>` HTML element with the `caption` class:

```
<span class="spacer"></span>
<mat-button-toggle-group #group="matButtonToggleGroup">
  <mat-button-toggle value="card" checked>
    <mat-icon>list</mat-icon>
  </mat-button-toggle>
  <mat-button-toggle value="table">
    <mat-icon>grid_on</mat-icon>
  </mat-button-toggle>
</mat-button-toggle-group>
```

In the preceding snippet, we use the `<mat-button-toggle-group>` element to create two toggle buttons side by side. The instance of the button toggle group is assigned to the group template reference variable so we can access it later.

We declare toggle buttons using the `<mat-button-toggle>` element and setting an appropriate value. The `value` property will be set when we click on either button. We also have an icon for each toggle button to enhance UX while users interact with the product list.

4. Create a new `@if` block after the `<div>` HTML element with the `caption` class and move the `@for` block inside it:

```
@if (group.value === 'card') {
  @for (product of products | sort; track product.id) {
    <mat-card [routerLink]=[product.id]>
      <mat-card-header>
        <mat-card-title-group>
          <mat-card-title>{{ product.title }}</mat-card-title>
          <mat-card-subtitle>{{ product.category }}</mat-card-
        subtitle>
          <img mat-card-sm-image [src]=[product.image] />
        </mat-card-title-group>
      </mat-card-header>
    </mat-card>
  } @empty {
    <p>No products found!</p>
  }
}
```

According to the preceding snippet, the card representation of products will be displayed when the `value` property of the button toggle group is set to `card`.

5. Add the following `@else` block and move the data table component inside it to display the product list in tabular format when the second toggle button is clicked:

```
@else {
  <table mat-table [dataSource]=[products]>
    <ng-container matColumnDef="title">
      <th mat-header-cell *matHeaderCellDef>Title</th>
      <td mat-cell *matCellDef="let product">
```

```
        <a [routerLink]=[product.id]>{{ product.title }}</a>
      </td>
    </ng-container>
    <ng-container matColumnDef="price">
      <th mat-header-cell *matHeaderCellDef>Price</th>
      <td mat-cell *matCellDef="let product">{{ product.price | currency }}</td>
    </ng-container>
    <tr mat-header-row *matHeaderRowDef="columnNames"></tr>
    <tr mat-row *matRowDef="let row; columns: columnNames;"></tr>
  </table>
}
```

6. Run the `ng serve` command to start the application and verify that the card representation is initially displayed.

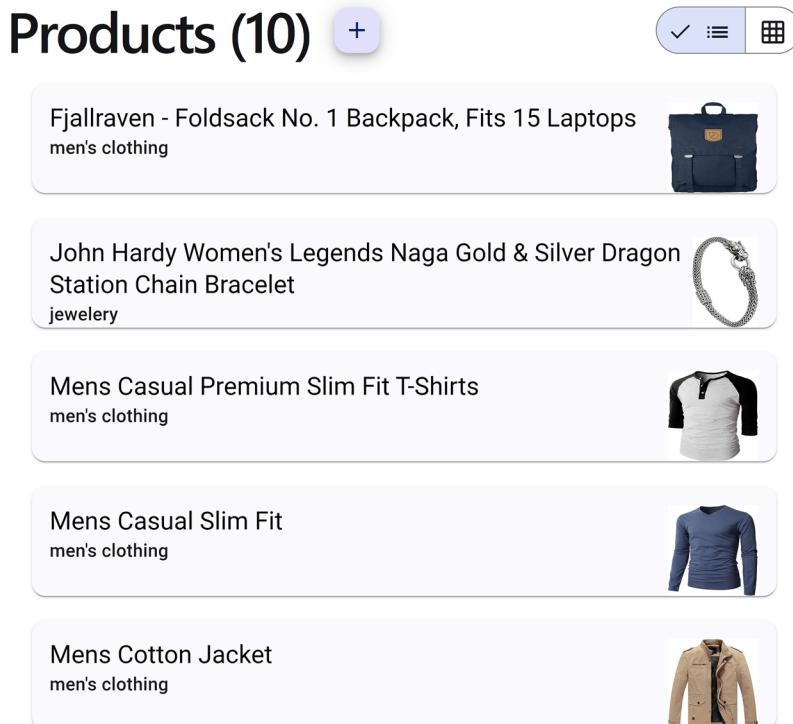


Figure 12.13: Product list

7. Click the second toggle button and verify that the products are now displayed in tabular format.

In this section, we learned how to display the product list in a tabular format. We also used toggle buttons to switch from the card view to the tabular one.

In the following section, we will learn how to use popups and overlays to provide additional information to the users.

Popups and overlays

There are different ways to capture the user's attention in a web application. One of them is to show a pop-up dialog over the content of the page and prompt the user to act accordingly. Another way is displaying information as a notification in different parts of the page.

Angular Material offers three different components for handling such cases:

- **Dialog:** A modal pop-up dialog that displays itself on top of the page content.
- **Badge:** A small circled indication to update the status of a UI element.
- **Snackbar:** An information message displayed at the bottom of a page that is visible briefly. Its purpose is to notify the user of the result of an action, such as saving a form.

We will learn how to use the preceding components in our e-shop application, starting with how to create a simple modal dialog.

Creating a confirmation dialog

The dialog component is quite powerful and can easily be customized and configured. It is an ordinary Angular component with custom directives that force it to behave like a dialog. To explore the capabilities of the Angular Material dialog, we will use a confirmation dialog in the checkout guard to notify users about remaining items in their shopping carts:

1. Run the following Angular CLI command to create a new Angular component:

```
ng generate component checkout
```

The preceding command will create an Angular component that will host our dialog.

2. Open the `checkout.component.ts` file and add the following import statements:

```
import { MatButton } from '@angular/material/button';
import { MatDialogModule } from '@angular/material/dialog';
```



The Angular Material dialog component consists of many other components and directives. We choose to import the whole Angular module because it would not be convenient to import them all individually.

3. Add the preceding imported classes in the `imports` array of the `@Component` decorator:

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

4. Open the `checkout.component.html` file and replace its content with the following HTML template:

```
<h1 mat-dialog-title>Cart Checkout</h1>
<mat-dialog-content>
  <span>You have pending items in your cart. Do you want to
  continue?</span>
</mat-dialog-content>
<mat-dialog-actions>
  <button mat-raised-button>Yes</button>
  <button mat-button>No</button>
</mat-dialog-actions>
```

The component template contains various directives and elements that belong to the Angular Material dialog component. The `mat-dialog-title` directive defines the title of the dialog, and the `<mat-dialog-content>` is the actual content. The `<mat-dialog-actions>` element defines the actions the dialog can perform and usually wraps button elements.

5. A dialog must be triggered to be displayed on a page. Open the `checkout.guard.ts` file and add the following `import` statements:

```
import { MatDialog } from '@angular/material/dialog';
import { CheckoutComponent } from './checkout/checkout.component';
```

6. Inject the `MatDialog` service in the body of the `checkoutGuard` function:

```
const dialog = inject(MatDialog);
```

7. Modify the assignment of the confirmation variable as follows:

```
if (cartService.cart) {  
  const confirmation = dialog.open(CheckoutComponent).afterClosed();  
  return confirmation;  
}
```

In the preceding snippet, we use the `MatDialog` service to display the checkout component. The `MatDialog` service accepts the type of component class representing the dialog as a parameter.

The `open` method of the `MatDialog` service returns an `afterClosed` observable property, which will notify us when the dialog closes. The observable emits any value that is sent back from the dialog.



Later in the chapter, we will learn how to return a boolean value from the dialog component that matches the type returned by the `CanDeactivateFn` function.

We can now verify that the dialog component works as expected by executing the following steps:

1. Run the application using the `ng serve` command and navigate to `http://localhost:4200`.
2. Log in to the application.
3. Select a product from the list and add it to the shopping cart.
4. Repeat the preceding step to add more products to the cart.
5. Navigate to the shopping cart and then click the back button of the browser or any of the application links to leave the cart. The following dialog will be displayed on the screen:

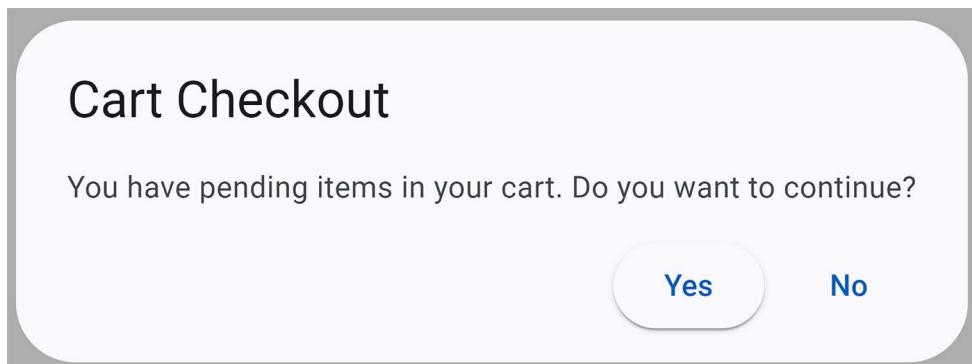


Figure 12.14: Checkout dialog component

We could improve the application's UX further by displaying information on the dialog about the number of items we have added to the shopping cart. In the following section, we will learn how to pass data in the dialog and display the number of shopping cart items.

Configuring dialogs

In a real-world scenario, you will probably need to create a reusable component to display a dialog in an Angular project. The component may end up in an Angular library as a package. Therefore, you should configure the dialog component to accept data dynamically.

In the current Angular project, we would like to display the number of products that we have added to the shopping cart:

1. Open the `checkout.component.ts` file and modify the `import` statements as follows:

```
import { Component, inject } from '@angular/core';
import { MatButton } from '@angular/material/button';
import { MatDialogModule, MAT_DIALOG_DATA } from '@angular/material/dialog';
```

2. Inject `MAT_DIALOG_DATA` in the `CheckoutComponent` class in the following way:

```
export class CheckoutComponent {
  data = inject(MAT_DIALOG_DATA);
}
```

The `MAT_DIALOG_DATA` is an injection token that enables us to pass arbitrary data to the dialog component. The `data` variable will contain any data we pass to the dialog when we call its `open` method.

3. Open the `checkout.component.html` file and add the `data` property to the inner text of the `` HTML element:

```
<span>
  You have {{ data }} pending items in your cart.
  Do you want to continue?
</span>
```

4. Open the `checkout.guard.ts` file and set the `data` property in the dialog configuration object, which is the second parameter of the `open` method:

```
const confirmation = dialog.open(
  CheckoutComponent,
```

```
{ data: cartService.cart.products.length }  
).afterClosed();
```

5. If we try to leave the cart page while running the application, we will get a dialog similar to the following:

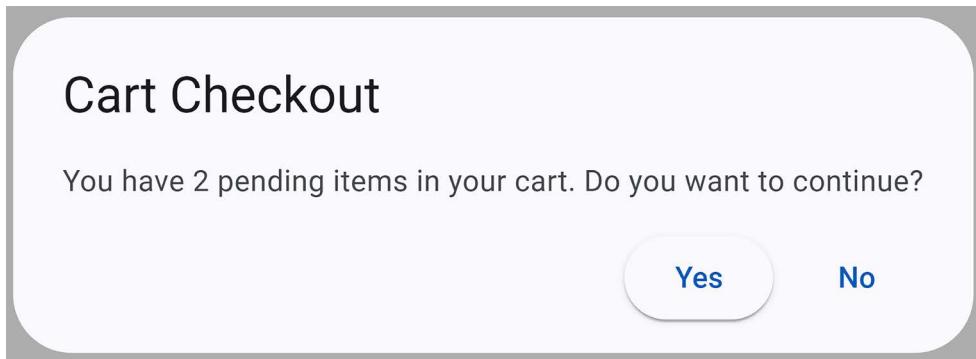


Figure 12.15: Checkout dialog component with custom data

The buttons of the dialog component don't do anything specific yet. In the following section, we will learn how to configure them and return data to the guard.

Getting data from dialogs

The Angular Material dialog module exposes the `mat-dialog-close` directive that we can use to configure which button will close the dialog. Open the `checkout.component.html` file and add the `mat-dialog-close` directive to both buttons:

```
<mat-dialog-actions>  
  <button mat-raised-button mat-dialog-close>Yes</button>  
  <button mat-button [mat-dialog-close]="false">No</button>  
</mat-dialog-actions>
```

In the preceding snippet, we use the `mat-dialog-close` directive in two ways:

- Without passing a value in the Yes button, the dialog will return `true` as a default value, allowing the guard to navigate away from the shopping cart page.
- With property binding in the No button, we pass `false` as a value to cancel the navigation from the guard.

Execute the following steps to verify that the dialog behavior is correct:

1. Run the `ng serve` command to start the application and navigate to `http://localhost:4200`.
2. Log in to the application.
3. Select a product from the list and add it to the cart.
4. Click the **My Cart** link to navigate to the shopping cart.
5. Click the **Products** link, select **No** in the checkout dialog, and verify that the application stays on the shopping cart page.
6. Click the **Products** link again, select **Yes** in the dialog, and you should navigate to the product list.

Dialogs are a great feature of Angular Material that can give your applications powerful capabilities. In the following section, we will explore the badge and snackbar components for notifying the user when a product is added to the shopping cart.

Displaying user notifications

The Angular Material library enforces patterns and behaviors that improve the application's UX. One aspect of the application UX concerns providing notifications to users upon specific actions. Angular Material gives us the badge and the snackbar components we can use in this case.

Applying badges

The badge component is a circle positioned on top of another element and usually displays a number. We will learn how to apply badges by displaying the number of shopping cart items in the **My Cart** application link:

1. Open the `app.component.ts` file and add the following `import` statements:

```
import { MatBadge } from '@angular/material/badge';
import { CartService } from './cart.service';
```

The `MatBadge` class exports the badge component. The `CartService` class will provide us with the number of items in the shopping cart.

2. Add the `MatBadge` class in the `imports` array of the `@Component` decorator:

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

```
    RouterLink,
    CopyrightDirective,
    AuthComponent,
    MatToolbarRow,
    MatToolbar,
    MatButton,
    MatBadge
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

3. Inject the `CartService` class in the `AppComponent` class:

```
cartService = inject(CartService);
```

4. Open the `app.component.html` file and add the `matBadge` directive to the `My Cart` button:

```
<button
  mat-button
  routerLink="/cart"
  [matBadge]="cartService.cart?.products?.length">
  My Cart
</button>
```

In the preceding snippet, the `matBadge` directive indicates the number displayed in the badge. In this case, we bind it with the `length` of the `products` array that exists in the current shopping cart.

5. Open the `app.component.css` file and add the following CSS style:

```
button {
  margin: 5px;
}
```

The preceding style will add space around each application link so that buttons do not overlap with the badge component.

6. Run the `ng serve` command to start the application and add some products to the shopping cart. Notice that the badge icon updates its value when products are added to the cart; here's an example:



Figure 12.16: Badge component

Applying a snackbar

Another good UX pattern when we work with CRUD applications is to display a notification when an action has been completed. We can apply such a pattern by displaying a notification when a product is added to the shopping cart. We will use the snackbar component of the Angular Material to show the notification:

1. Open the `product-detail.component.ts` file and add the following import statement:

```
import { MatSnackBarModule, MatSnackBar } from '@angular/material/snack-bar';
```

The snackbar is not an actual Angular component like all the Angular Material components we have seen. It is an Angular service named `MatSnackBar` and can be used by importing the `MatSnackBarModule` class into our components.

2. Add the `MatSnackBarModule` class in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-detail',
  imports: [
    CommonModule,
    FormsModule,
    PriceMaximumDirective,
    MatButton,
    MatInput,
    MatFormField,
    MatError,
    MatIcon,
    MatSuffix,
    MatIconButton,
    MatChipSet,
    MatChip,
    MatSnackBarModule
  ],
})
```

```
        templateUrl: './product-detail.component.html',
        styleUrls: ['./product-detail.component.css']
    })
```

3. Inject the MatSnackBar service into the constructor of the ProductDetailComponent class:

```
constructor(
    private productService: ProductsService,
    public authService: AuthService,
    private route: ActivatedRoute,
    private router: Router,
    private cartService: CartService,
    private snackbar: MatSnackBar
) {}
```

4. Modify the addToCart method to display a snackbar when the product is added to the cart:

```
addToCart(id: number) {
    this.cartService.addProduct(id).subscribe(() => {
        this.snackbar.open('Product added to cart!', undefined, {
            duration: 1000
        });
    });
}
```

In the preceding method, we use the open method of the MatSnackBar service to display a snackbar. The open method accepts three parameters: the message we want to display, any action we want to take when the snackbar is dismissed, and a configuration object. The configuration object enables us to set various options, such as the duration for which the snackbar will be visible in milliseconds.



We do not pass a parameter for the action because we do not want to react when the snackbar is dismissed.

5. Run the `ng serve` command to start the application and select a product from the list.
6. Ensure you are logged in and click the **Add to cart** button. The following notification message will be displayed at the bottom of the page:

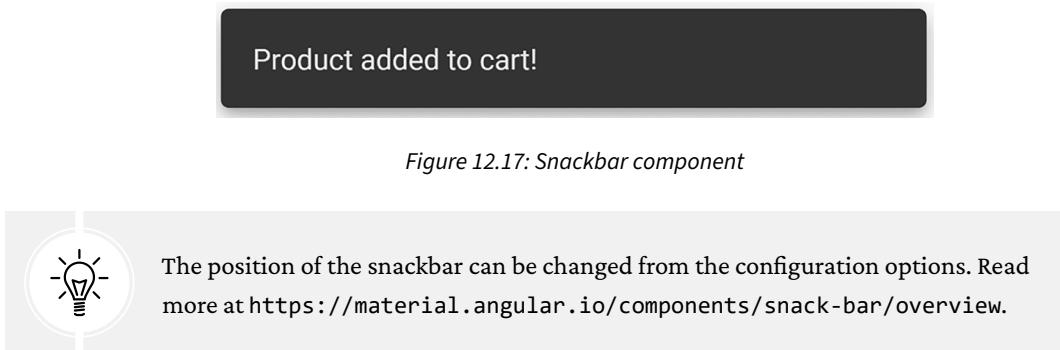
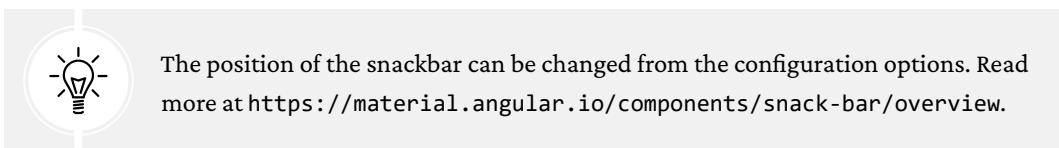


Figure 12.17: Snackbar component



The position of the snackbar can be changed from the configuration options. Read more at <https://material.angular.io/components/snack-bar/overview>.

In this section, we learned to use pop-up models and notification overlays to enhance the application UX and provide a great workflow to our users.

Summary

In this chapter, we looked at the basics of the Material Design system. We put most of our focus on Angular Material, the Material Design implementation meant for Angular, and how it consists of different components. We looked at a hands-on explanation of how to install it, set it up, and use some of its core components and themes.

Hopefully, you will have read this chapter and found that you now grasp Material Design in general and Angular Material in particular and can determine whether it is a good match for your next Angular application.

Web applications must be testable to ensure they are functional and in accordance with the application requirements. In the next chapter, we will learn how to apply different testing techniques in Angular applications.

13

Unit Testing Angular Applications

In the previous chapters, we went through many aspects of how to build an Angular enterprise application from scratch. But how can we ensure that an application can be maintained in the future without much hassle? A comprehensive automated testing layer can become our lifeline once our application begins to scale up and we have to mitigate the impact of bugs.

Testing, specifically unit testing, is meant to be carried out by the developer as the project is being developed. Now that our knowledge of the framework is mature, we will briefly cover all the intricacies of unit testing an Angular application in this chapter including the use of testing tools..



For simplicity, the examples in this chapter are not related to the e-shop application that we have built throughout the book.

In more detail, we will learn about the following:

- Why do we need unit tests?
- The anatomy of a unit test
- Introducing unit tests in Angular
- Testing components
- Testing services

- Testing pipes
- Testing directives
- Testing forms
- Testing the router

Technical requirements

The chapter contains various code samples to walk you through the concept of unit testing in Angular. You can find the related source code in the ch13 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>.

Why do we need unit tests?

In this section, we will learn what unit tests are and why they are useful in web development.



You can skip to the next section if you're familiar with unit testing and test-driven development.

Unit tests are part of an engineering philosophy for efficient and agile development processes. They add a layer of automated testing to the application code before it is developed. The core concept is that a piece of code is accompanied by its test, both of which are built by the developer who works on that code. First, we design the test against the feature we want to deliver, checking the accuracy of its output and behavior. Since the feature is still not implemented, the test will fail, so the developer's job is to build the feature to pass the test.

Unit testing is quite controversial. While test-driven development is beneficial for ensuring code quality and maintenance over time, not everybody undertakes unit testing in their daily development workflow.

Building tests as we develop our code can sometimes feel like a burden, especially when the test results become larger than the functionality it aims to test. However, the arguments in favor of testing outnumber the arguments against it:

- Building tests contributes to better code design. Our code must conform to the test requirements and not vice versa. If we try to test an existing piece of code and find ourselves blocked at some point, the chances are that the code is not well designed and requires some rethinking. On the other hand, building testable features can help with the early detection of side effects.

- Refactoring tested code is a lifeline against introducing bugs in later stages. Development is meant to evolve with time, and the risk of introducing a bug with every refactor is high. Unit tests are an excellent way to ensure we catch bugs early, either when introducing new features or updating existing ones.
- Building tests is an excellent way to document our code. It becomes a priceless resource when someone unfamiliar with the code base takes over the development endeavor.

These are only a few arguments, but you can find countless resources on the web about the benefits of testing your code. If you do not feel convinced yet, give it a try; otherwise, let's continue with our journey and look at the overall form of a unit test.

The anatomy of a unit test

There are many different ways to test a piece of code. In this chapter, we will look at the anatomy of a unit test—the separate parts it's made of.

To test any code, we need a framework for writing the test and a runner to run it on. In this section, we will focus on the test framework. The test framework should provide utility functions for building test suites containing one or several test specs. As a result, unit testing involves the following concepts:

- **Test suite:** A suite that creates a logical grouping for many tests. A suite, for example, can contain all the tests for a specific feature.
- **Test spec:** The actual unit test.

We will use **Jasmine** in this chapter, a popular test framework that is also used by default in Angular CLI projects. This is what a unit test looks like in Jasmine:

```
describe('Calculator', () => {
  it('should add two numbers', () => {
    expect(1+1).toBe(2);
  });
});
```

The `describe` method defines the test suite and accepts a name and an arrow function as parameters. The arrow function is the body of the test suite and contains several unit tests. The `it` method defines a single unit test. It accepts a name and an arrow function as parameters.

Each test spec validates a specific functionality of the feature described in the suite name and declares one or several expectations in its body. Each expectation takes a value, called the **expected** value, which is compared against an **actual** value using a **matcher** function. The function checks whether the expected and actual values match accordingly, which is called an **assertion**. The test framework passes or fails the spec depending on the result of such assertions. In the previous example, `1+1` will return the actual value that is supposed to match the expected value, `2`, declared in the `toBe` matcher function.



The Jasmine framework contains various matcher functions according to user-specific needs, as we will see later in the chapter.

Suppose the previous code contains another mathematical operation that must be tested. It would make sense to group both operations under the `Calculator` suite, as follows:

```
describe('Calculator', () => {
  it('should add two numbers', () => {
    expect(1+1).toBe(2);
  });

  it('should subtract two numbers', () => {
    expect(1-1).toBe(0);
  });
});
```

So far, we have learned about test suites and how to use them to group tests according to their functionality. Furthermore, we have learned about invoking the code we want to test and affirming that it does what it should do. However, more concepts are involved in unit tests that are worth knowing about, namely, the `setup` and `teardown` functionalities.

A setup functionality prepares your code before you start running the tests. It's a way to keep your code clean by focusing on invoking the code and checking the assertions. A teardown functionality is the opposite. It is responsible for tearing down what we initially set up, involving activities such as cleaning up resources. Let's see what this looks like in practice with a code example:

```
describe('Calculator', () => {
  let total: number;

  beforeEach(() => total = 1);
```

```
it('should add two numbers', () => {
    total = total + 1;
    expect(total).toBe(2);
});

it('should subtract two numbers', () => {
    total = total - 1;
    expect(total).toBe(0);
});

afterEach(() => total = 0);
});
```

The `beforeEach` method is used for the setup functionality and runs before every unit test. In this example, we set the value of the `total` variable to 1 before each test. The `afterEach` method is used to run teardown logic. After each test, we reset the value of the `total` variable to 0.

It is evident that the test only has to care about invoking application code and asserting the outcome, which makes tests cleaner; however, tests tend to have much more setup in a real-world application. Most importantly, the `beforeEach` method tends to make it easier to add new tests, which is great. We want well-tested code; the easier it is to write and maintain such code, the better for our software.

Now that we have covered the basics of a unit test, let's see how we can implement them in the context of the Angular framework.

Introducing unit tests in Angular

In the previous section, we familiarized ourselves with unit testing and its general concepts, such as test suites, test specs, and assertions. It is time to venture into unit testing with Angular, armed with that knowledge. Before we start writing tests for Angular, though, let's have a look at the tooling that the Angular framework and the Angular CLI provide us with:

- **Jasmine:** We have already learned that this is the testing framework.
- **Karma:** The test runner for running our unit tests.
- **Angular testing utilities:** A set of helper methods that assist us in setting up our unit tests and writing our assertions in the context of the Angular framework.



When we use the Angular CLI, we do not have to do anything to configure Jasmine and Karma in an Angular application. Unit testing works out of the box when we create a new Angular CLI project. Most of the time, we will interact with the Angular testing utilities.

Angular testing utilities help us to create a testing environment that makes writing tests for our Angular artifacts easy. It consists of the `TestBed` class and various helper methods in the `@angular/core/testing` namespace. As this chapter progresses, we will learn what these are and how they can help us test various artifacts. For now, let's have a look at the most commonly used concepts so that you are familiar with them when we look at them in more detail later on:

- `TestBed`: A class that creates a testing module. We attach an Angular artifact to this testing module when we test it. The `TestBed` class contains the `configureTestingModule` method we use to set up the test module as needed.
- `ComponentFixture`: A wrapper class around an Angular component instance. It allows us to interact with the component and its corresponding HTML element.
- `DebugElement`: A wrapper around the DOM element of the component. It is an abstraction that operates cross-platform so that our tests are platform-independent.

Now that we know our testing environment and the frameworks and libraries used, we can start writing our first unit tests in Angular.



All the examples described in this chapter have been created in a new Angular CLI project.

We will embark on this great journey from the most fundamental building block in Angular, the component.

Testing components

You may have noticed that whenever we used the Angular CLI to scaffold a new Angular application or generate an Angular artifact, it created some test files for us.

Test files in the Angular CLI contain the word `spec` in their filename. The filename of a test is the same as the Angular artifact it is testing, followed by the suffix `.spec.ts`. For example, the test file for the main component of an Angular application is `app.component.spec.ts` and it resides in the same path as the component file.



We should consider an Angular artifact and its corresponding test one thing. When we change the logic of the artifact, we may need to modify the unit test as well. Placing unit test files with their Angular artifacts makes it easier for us to remember and edit them. It also helps us when we need to refactor our code, such as moving artifacts (not forgetting to move the unit test).

When we scaffold a new Angular application, the Angular CLI automatically creates a test for the main component, `AppComponent`. At the beginning of the file, there is a `beforeEach` statement that is used for setup purposes:

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [AppComponent],
  }).compileComponents();
});
```

It uses the `configureTestingModule` method of the `TestBed` class and passes an object as a parameter.

We can specify an `imports` array that contains the component we want to test. Additionally, we can define teardown options using the `teardown` property.

The `teardown` property contains an object of the `ModuleTeardownOptions` type that can set the following properties:

- `destroyAfterEach`: It creates a new instance of the module at each test to eliminate bugs caused by the incomplete cleanup of HTML elements.
- `rethrowErrors`: It throws any errors that occur when the module is destroyed.

Finally, we call the `compileComponents` method to compile the TypeScript class and the HTML template of our component.

The first unit test verifies whether we can create a new instance of `AppComponent` using the `createComponent` method:

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
```

```
    expect(app).toBeTruthy();
});
```

The result of the `createComponent` method is a `ComponentFixture` instance of the `AppComponent` type that can give us the component instance using the `componentInstance` property. We also use the `toBeTruthy` matcher function to check whether the resulting instance is valid.

As soon as we have access to the component instance, we can query any of its public properties and methods:

```
it(`should have the 'my-app' title`, () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.title).toEqual('my-app');
});
```

In the previous test, we check whether the `title` component property is set to `my-app` using another matcher function, `toEqual`.



The value of the `title` component property in a new Angular application will be the name you passed in the `ng new` command while creating the application.

As we have learned, a component consists of a TypeScript class and a template file. So, testing it only from the class perspective, as in the previous test, is not sufficient. We should also test whether the class interacts correctly with the DOM:

```
it('should render title', () => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.nativeElement as HTMLElement;
  expect(compiled.querySelector('h1')?.textContent).toContain('Hello, my-
app');
});
```



Many developers favor class testing over DOM testing and rely on **end-to-end (E2E)** testing, which is slower and performs poorly. E2E tests often validate the integration of an application with a backend API and are easy to break. Thus, performing DOM unit testing in your Angular applications is recommended.

In the preceding test, we create a component and call the `detectChanges` method of the `ComponentFixture`. The `detectChanges` method triggers the Angular change detection mechanism, forcing the data bindings to be updated. It executes the `ngOnInit` life cycle event of the component the first time it is called and the `ngOnChanges` in subsequent calls so that we can query the DOM element of the component using the `nativeElement` property. In this example, we check the `textContent` of the HTML element corresponding to the `title` property.

To run tests, we use the `ng test` command of the Angular CLI. It will start the Karma test runner, fetch all unit test files, execute them, and open a browser to display the results of each test. The Angular CLI uses the Google Chrome browser by default. The output will look like this:

```
AppComponent
• should render title
• should create the app
• should have the 'my-app' title
```

Figure 13.1: Test execution output

In the previous figure, we can see the result of each test at the top of the page. We can also see how Karma visually groups each test by suite. In our case, the only test suite is `AppComponent`.

Now, let's make one of our tests fail. Open the `app.component.ts` file, change the value of the `title` property to `my-new-app`, and save the file. Karma will re-execute our tests and display the results on the page:

```
AppComponent > should have the 'my-app' title
Expected 'my-new-app' to equal 'my-app'.
at <Jasmine>
at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack:/src/app/app.component.spec.ts:20:23)
at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:368:26)
at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:273:39)
at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:367:52)

AppComponent > should render title
Expected 'Hello, my-new-app' to contain 'Hello, my-app'.
at <Jasmine>
at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack:/src/app/app.component.spec.ts:27:55)
at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:368:26)
at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:273:39)
at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:367:52)
```

Figure 13.2: Test failure



Karma runs in **watch mode**, so we do not need to execute the Angular CLI test command every time we make a change.

Sometimes, reading the output of tests in the browser is not very convenient. Alternatively, we can inspect the console window that we used to run the `ng test` command, which contains a trimmed version of the test results:

```
Executed 3 of 3 SUCCESS (0.117 secs / 0.044 secs)
TOTAL: 3 SUCCESS
```

We've gained quite a lot of insight just by looking at the test of `AppComponent` that the Angular CLI automatically created for us. In the following section, we will look at a more advanced scenario for testing a component with dependencies.

Testing with dependencies

In a real-world scenario, components are not usually as simple as the main component. They will almost certainly be dependent on one or more services. They will also possibly contain other child components in their template.

We have different ways of dealing with testing in such situations. One thing is clear: if we are testing the component, we should not test the service or its child components. So, when we set up such a test, the dependency should not be the real class. There are different ways of dealing with that when it comes to unit testing; no solution is strictly better than another:

- **Stubbing:** A method that instructs the dependency injector to inject a stub of the dependency that we provide instead of the real class.
- **Spying:** A method that injects the actual dependency but attaches a spy to the method that we call in our component. We can then either return mock data or let the method call through.



Using stubbing over spying is preferable when a dependency is complicated. Some services inject other services, so using the real dependency in a test requires you to compensate for other dependencies. It is also the preferred method when the component we want to test contains child components in its template.

Regardless of the approach, we ensure that the test does not perform unintended actions, such as accessing the filesystem or attempting to communicate via HTTP; we are testing the component in complete isolation.

Replacing the dependency with a stub

Replacing a dependency with a stub means that we completely replace the dependency with a fake one.

We can create a fake dependency in the following ways:

- Create a constant variable or class that contains properties and methods of the real dependency.
- Create a mock definition of the actual class of the dependency.

The approaches are not so different. In this section, we will look at the first one as it is most common in Angular development. Feel free to explore the second one at your own pace.

Consider the following `stub.component.ts` component file:

```
import { Component, OnInit } from '@angular/core';
import { StubService } from '../stub.service';

@Component({
  selector: 'app-stub',
  template: '<span>{{ msg }}</span>'
})
export class StubComponent implements OnInit {
  msg = '';

  constructor(private stubService: StubService) {}

  ngOnInit(): void {
    this.msg = this.stubService.isBusy
      ? this.stubService.name + ' is on mission'
      : this.stubService.name + ' is available';
  }
}
```

It injects `StubService`, which contains two public properties. Providing a stub for this service in tests is pretty straightforward, as shown in the following example:

```
const serviceStub: Partial<StubService> = {  
  name: 'Boothstomper'  
};
```

We have declared the service as `Partial` because we want only to set the `name` property initially. We can now use the object literal syntax to inject the stub service in our testing module:

```
await TestBed.configureTestingModule({  
  imports: [StubComponent],  
  providers: [  
    { provide: StubService, useValue: serviceStub }  
  ]  
})  
.compileComponents();
```

The `msg` component property relies on the value of the `isBusy` service property. Therefore, we need to get a reference to the service in the test suite and provide alternate values for this property in each test. We can get the injected instance of `StubService` using the `inject` method of the `TestBed` class:

```
describe('status', () => {  
  let service: StubService;  
  
  beforeEach(() => {  
    service = TestBed.inject(StubService);  
  })  
});
```



We pass the real `StubService` as a parameter to the `inject` method, not the stubbed version we created. Modifying the value of the stub will not affect the injected service since our component uses an instance of the real service. The `inject` method asks the root injector of the application for the requested service. If the service was provided from the component injector, we would need to get it from the component injector using `fixture.debugElement.injector.get(StubService)`.

We can now write our tests to check whether the `msg` component property behaves correctly during data binding:

```
describe('status', () => {
  let service: StubService;
  let msgDisplay: HTMLElement;

  beforeEach(() => {
    service = TestBed.inject(StubService);
    msgDisplay = fixture.nativeElement.querySelector('span');
  })

  it('should be on a mission', () => {
    service.isBusy = true;
    fixture.detectChanges();
    expect(msgDisplay.textContent).toContain('is on mission');
  });

  it('should be available', () => {
    service.isBusy = false;
    fixture.detectChanges();
    expect(msgDisplay.textContent).toContain('is available');
  });
});
```



We have removed the `fixture.detectChanges` line from the `beforeEach` statement because we want to trigger change detection in our tests separately.

Stubbing a dependency is not always viable, especially when the root injector does not provide it. A service can be provided at the component injector level. Providing a stub using the process we looked at earlier doesn't have any effect. To tackle such a scenario, we can use the `overrideComponent` method of the `TestBed` class:

```
await TestBed.configureTestingModule({
  imports: [StubComponent],
  providers: [
    { provide: StubService, useValue: serviceStub }
```

```
        ]
    })
    .overrideComponent(StubComponent, {
      set: {
        providers: [
          { provide: StubService, useValue: serviceStub }
        ]
      }
    })
  )
.compileComponents();
```

The `overrideComponent` method accepts two parameters: the type of component that provides the service and an override metadata object. The metadata object contains the `set` property, which provides services to the component.

Suppose that the component we want to test contains a child component in its template, such as:

```
@Component({
  selector: 'app-stub',
  template: `
    <span>{{ msg }}</span>
    <app-child></app-child>
  `
})
```

In the preceding case, when we tested the `StubComponent`, we also needed to import the TypeScript class of the `<app-child>` component when configuring the testing module:

```
await TestBed.configureTestingModule({
  imports: [StubComponent],
  providers: [
    { provide: StubService, useValue: serviceStub }
  ],
  imports: [ChildComponent]
})
```

The `ChildComponent` class may have other dependencies as well. Providing stubs for those dependencies is not viable because it is not the responsibility of the component under test. Instead, we can create a stub TypeScript class for the component and import it when configuring the testing module:

```
@Component({ selector: 'app-child', template: '' })
class ChildStubComponent {}
```

In the preceding snippet, we passed an empty array in the `template` property of the component because we are not interested in the internal implementation of the child component.



If the child component contains properties and methods that are used while testing the parent component, we need to define them as well in the `ChildStubComponent`.

Alternatively, to provide a stub of the component, we can pass the `NO_ERRORS_SCHEMA` from the `@angular/core` npm package while configuring the testing module:

```
await TestBed.configureTestingModule({
  imports: [StubComponent],
  providers: [
    { provide: StubService, useValue: serviceStub },
  ],
  schemas: [NO_ERRORS_SCHEMA]
})
```

The preceding snippet instructs Angular to ignore any components that have not been imported into the testing module.

Stubbing a dependency is very simple, but it is not always possible, as we will see in the following section.

Spying on the dependency method

Using a stub is not the only way to isolate logic in a unit test. We don't have to replace the entire dependency—only the parts our component uses. Replacing certain parts means we point out specific methods on the dependency and assign a spy to them. A spy can answer what you want, but you can also see how many times it was called and with what arguments. So, a spy gives you much more information about what is happening.

There are two ways to set up a spy in a dependency:

- Inject the actual dependency and spy on its methods.
- Use the Jasmine `createSpyObj` method to create a fake dependency instance. We can then spy on the methods of this dependency as we would with the real one.

The first case is most common in Angular development. Let's see how to set it up. Consider the following `spy.component.ts` file, which uses the `Title` service of the Angular framework:

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

@Component({
  selector: 'app-spy',
  template: '{{ caption }}'
})
export class SpyComponent implements OnInit {
  caption = '';

  constructor(private title: Title) {}

  ngOnInit(): void {
    this.title.setTitle('My Angular app');
    this.caption = this.title.getTitle();
  }
}
```



The `Title` service interacts with the title of the main HTML document in an Angular application.

We do not have any control over the `Title` service since it is built into the framework. It may have dependencies that we do not know about. Spying on its methods is the easiest and safest way to use it in our tests. We inject it in the testing module using the `providers` array and then use it in our test, such as:

```
it('should set the title', () => {
  const title = TestBed.inject(Title);
  const spy = spyOn(title, 'setTitle');
  component.ngOnInit();
  expect(spy).toHaveBeenCalledWith('My Angular app');
});
```

We use the Jasmine `spyOn` method, which accepts two parameters: the object and its specific method to spy. We used it before calling the `ngOnInit` component method to attach the spy before triggering the change detection mechanism. The `expect` statement validates that the `setTitle` method was called with the correct arguments.

Our component also uses the `getTitle` method to get the document title. We can spy directly on that method and return mock data:

1. First, we need to define the `Title` service as a spy object and initialize it by passing two parameters—the name of the service and an array of the method names that the component currently uses:

```
const titleSpy = jasmine.createSpyObj('Title', [
  'getTitle', 'setTitle'
]);
```

2. Then we attach a spy to the `getTitle` method and return a custom title using the Jasmine `returnValue` method:

```
titleSpy.getTitle.and.returnValue('My title');
```

3. Finally, we add the `titleSpy` variable in the providers array of the testing module:

```
await TestBed.configureTestingModule({
  imports: [SpyComponent],
  providers: [
    { provide: Title, useValue: titleSpy }
  ]
})
.compileComponents();
```

The resulting test should look like the following:

```
it('should get the title', async () => {
  const titleSpy = jasmine.createSpyObj('Title', [
    'getTitle', 'setTitle'
]);
  titleSpy.getTitle.and.returnValue('My title');

  await TestBed.configureTestingModule({
    imports: [SpyComponent],
```

```
    providers: [
      { provide: Title, useValue: titleSpy }
    ]
  })
.compileComponents();

const fixture = TestBed.createComponent(SpyComponent);
fixture.detectChanges();

expect(fixture.nativeElement.textContent).toContain('My title');
});
```

Very few services are well behaved and straightforward, such as the `Title` service, in the sense that they are synchronous. Most of the time, they are asynchronous and can return observables or promises. In the following section, we will learn how to test asynchronous dependencies.

Testing asynchronous services

Angular testing utilities provide two artifacts to tackle asynchronous testing scenarios:

- `waitForAsync`: An asynchronous approach to unit test services. It is combined with the `whenStable` method of the `ComponentFixture` class.
- `fakeAsync`: A synchronous approach to unit test services. It is used in combination with the `tick` function.

Both approaches provide roughly the same functionality; they only differ in how we use them. Let's see how we can use each by looking at an example.

Consider the following `async.component.ts` file:

```
import { AsyncPipe } from '@angular/common';
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { AsyncService } from '../async.service';

@Component({
  selector: 'app-async',
  imports: [AsyncPipe],
  template: `
    @for(item of items$ | async; track item) {
```

```
        <p>{{ item }}</p>
    }
  )
}

export class AsyncComponent implements OnInit {
  items$: Observable<string[]> | undefined;

  constructor(private asyncService: AsyncService) {}

  ngOnInit(): void {
    this.items$ = this.asyncService.getItems();
  }
}
```

It injects the `AsyncService` from the `async.service.ts` file and calls its `getItems` method inside the `ngOnInit` method. As we can see, the `getItems` method returns an observable of strings. It also introduces a slight delay so that the scenario looks asynchronous:

```
getItems(): Observable<string[]> {
  return of(items).pipe(delay(500));
}
```

The unit test queries the native element of the component and checks whether the value of the `items$` observable is displayed correctly:

```
it('should get data with waitForAsync', waitForAsync(async() => {
  fixture.detectChanges();
  await fixture.whenStable();
  fixture.detectChanges();

  const itemDisplay: HTMLElement[] = fixture.nativeElement.
  querySelectorAll('p');
  expect(itemDisplay.length).toBe(2);
}));
```

We wrap the test body inside the `waitForAsync` method and call the `detectChanges` method to trigger change detection. Furthermore, we call the `whenStable` method, which returns a promise that is resolved immediately when the `items$` observable is complete. When the promise is resolved, we call the `detectChanges` method again to trigger data binding and query the DOM accordingly.



The `whenStable` method is also used when we want to test a component that contains a template-driven form. The asynchronous nature of this method makes it preferable to use reactive forms in our Angular applications.

An alternative synchronous approach would be to use the `fakeAsync` method and write the same unit test as follows:

```
it('should get items with fakeAsync', fakeAsync(() => {
  fixture.detectChanges();
  tick(500);
  fixture.detectChanges();

  const itemDisplay: HTMLElement[] = fixture.nativeElement.
  querySelectorAll('p');
  expect(itemDisplay.length).toBe(2);
}));
```

In the previous snippet, we wrapped the test body in a `fakeAsync` method and replaced the `whenStable` method with the `tick` function. The `tick` function advances the time by 500 ms, which is the virtual delay we introduced in the `getItems` method of the `AsyncService`.

Testing components with asynchronous services can sometimes become a nightmare. Still, each of the described approaches can significantly help us in this task. However, components are not only about services but also input and output bindings. In the following section, we will learn how to test the public API of a component.

Testing with inputs and outputs

So far, we have learned how to test components with simple properties and tackle synchronous and asynchronous dependencies. But there is more to a component than that. As we learned in *Chapter 3, Structuring User Interfaces with Components*, a component has a public API consisting of inputs and outputs that should also be tested.

Since we want to test the public API of a component, it makes sense to test how it interacts when hosted from another component. Testing such a component can be done in two ways:

- We can verify that our input binding is correctly set.
- We can verify that our output binding triggers correctly and that what it emits is received.

Suppose that we have the following `bindings.component.ts` file with an input and output binding:

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

@Component({
  selector: 'app-bindings',
  template: `
    <p>{{ title() }}</p>
    <button (click)="liked.emit()">Like!</button>
  `
})
export class BindingsComponent {
  title = input('');
  liked = output();
}
```

Before we start writing our tests, we should create a test host component inside the `bindings.component.spec.ts` file that is going to use the component under test:

```
@Component({
  imports: [BindingsComponent],
  template: `
    <app-bindings [title]="testTitle" (liked)="isFavorite = true"></app-bindings>
  `
})
export class TestHostComponent {
  testTitle = 'My title';
  isFavorite = false;
}
```

In the setup phase, notice that the `ComponentFixture` is of the `TestHostComponent` type:

```
describe('BindingsComponent', () => {
  let component: TestHostComponent;
  let fixture: ComponentFixture<TestHostComponent>;
  let titleElement: HTMLElement;

  beforeEach(async () => {
```

```
await TestBed.configureTestingModule({
  imports: [TestHostComponent]
})
.compileComponents();

fixture = TestBed.createComponent(TestHostComponent);
component = fixture.componentInstance;
fixture.detectChanges();
});

it('should create', () => {
  expect(component).toBeTruthy();
});
});
```

Our unit tests will validate the behavior of `BindingsComponent` when interacting with `TestHostComponent`.

The first test checks whether the input binding to the `title` property has been applied correctly:

```
it('should display the title', () => {
  const titleDisplay: HTMLElement = fixture.nativeElement.
  querySelector('p');
  expect(titleDisplay.textContent).toEqual(component.titleLabel);
});
```

The second test validates whether the `isFavorite` property is wired up correctly with the `liked` output event:

```
it('should emit the liked event', () => {
  const button: HTMLButtonElement = fixture.nativeElement.
  querySelector('button');
  button.click();
  expect(component.isFavorite).toBeTrue();
});
```

In the previous test, we query the DOM for the `<button>` element using the `nativeElement` property of the `ComponentFixture` class. Then, we click on it for the output event to emit. Alternatively, we could have used the `debugElement` property to find the button and use its `triggerEventHandler` method to click on it:

```
it('should emit the liked event using debugElement', () => {
  const buttonDe = fixture.debugElement.query(By.css('button'));
  buttonDe.triggerEventHandler('click');
  expect(component.isFavorite).toBeTrue();
});
```

In the preceding test, we use the `query` method, which accepts a **predicate** function as a parameter. The predicate uses the `CSS` method of the `By` class to locate an element by its CSS selector.



As we learned in the *Introducing unit tests in Angular* section, the `debugElement` is framework agnostic. If you are sure that your tests will only run in a browser, you should go with the `nativeElement` property.

The `triggerEventHandler` method accepts the event name we want to trigger as a parameter; in this case, it is the `click` event.

We could have avoided a lot of code if we had only tested the `BindingsComponent`, which would still have been valid. But we would have missed the opportunity to test it as a real-world scenario. The public API of a component is intended to be used by other components, so we should test it in this way.

Currently, the button we use in the template of the `BindingsComponent` is a native HTML `<button>` element. If the button was an Angular Material button component, we could use an alternate approach for interacting with it, which is the topic of the following section.

Testing with a component harness

The Angular CDK library, the core of Angular Material, contains a set of utilities that allow a test to interact with a component over a public testing API. Angular CDK testing utilities enable us to access Angular Material components without relying on their internal implementation using a **component harness**.

The process of testing an Angular component using a harness consists of the following parts:

- `@angular/cdk/testing`: The npm package that contains infrastructure for interacting with a component harness.
- **Testing environment**: The environment in which the component harness test will be loaded. The Angular CDK contains a built-in testing environment for unit testing with Karma. It also provides a rich set of tools that allow developers to create custom testing environments.

- **Component harness:** A class that gives the developer access to the instance of a component in the browser DOM.

To learn how to use component harnesses, we will convert the `<button>` element of the `BindingsComponent` into an Angular Material button:

```
import { Component, input, output } from '@angular/core';
import { MatButton } from '@angular/material/button';

@Component({
  selector: 'app-bindings',
  imports: [MatButton],
  template: `
    <p>{{ title() }}</p>
    <button mat-button (click)="liked.emit()">Like!</button>
  `
})

```



The preceding snippet assumes that you have added the Angular Material library to the project that you are working on.

To start using a component harness from the Angular CDK, we need to import the following artifacts from the `@angular/cdk/testing` namespace:

```
import { TestbedHarnessEnvironment } from '@angular/cdk/testing/testbed';
import { MatButtonHarness } from '@angular/material/button/testing';
```

In the preceding snippet, we have added the following classes:

- `TestbedHarnessEnvironment`: Represents the testing environment for running unit tests with Karma.
- `MatButtonHarness`: The component harness for the Angular Material button component. Almost all components of the Angular Material library have a corresponding component harness that we can use.



If you are a component library author, the Angular CDK provides all the necessary tools for creating harnesses for your UI components.

After we have finished importing all the necessary artifacts, we can write our test:

```
it('should emit the liked event using harness', async () => {
  const loader = TestBedHarnessEnvironment.loader(fixture);
  const buttonHarness = await loader.getHarness(MatButtonHarness);
  await buttonHarness.click();
  expect(component.isFavorite).toBeTrue();
});
```

In the preceding test, the `loader` method of the testing environment accepts the `ComponentFixture` instance of the current component as a parameter and returns a `HarnessLoader` object. The abstraction that an Angular CDK harness provides is based on the concept that it operates on the component fixture, which is an abstraction layer on top of the actual DOM element.

We surround the body of the test inside an `async` function because component harnesses are promise-based. We use the `getHarness` method of the harness loader to load the specific harness for the button component. Finally, we call the `click` method of the button component harness to trigger the button click event.



We do not need to call the `detectChanges` method because the Angular CDK component harness triggers change detection automatically.

The component harness is a powerful Angular CDK tool that ensures we interact with components abstractly and safely during testing.

We have discussed many ways to test a component with a dependency. Now, it is time to learn how to test the dependency itself.

Testing services

As we learned in *Chapter 5, Managing Complex Tasks with Services*, a service can inject other services. Testing a standalone service is pretty straightforward: we get an instance from the injector and then start to query its public properties and methods.



We are only interested in testing the public API of a service, which is the interface that components and other artifacts use. Private properties and methods do not have any value when tested because they represent the internal implementation of the service.

There are two different types of testing that we can perform in a service:

- Testing synchronous and asynchronous operations, such as a method that returns a simple array or one that returns an observable
- Testing services with dependencies, such as a method that makes HTTP requests

In the following sections, we will go through each in more detail.

Testing synchronous/asynchronous methods

When we create an Angular service using the Angular CLI, it also creates a corresponding test file. Consider the following `async.service.spec.ts` file, which is the test file for the `AsyncService` we used earlier:

```
import { TestBed } from '@angular/core/testing';

import { AsyncService } from './async.service';

describe('AsyncService', () => {
  let service: AsyncService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  
    service = TestBed.inject(AsyncService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

The `AsyncService` is not dependent on anything. It is also provided with the root injector of the Angular application, so it passes an empty object to the `configureTestingModule` method. We can get an instance of the service we test using the `inject` method of the `TestBed` class.

The first test that we can write is pretty straightforward as it calls the `setItems` method and inspects its result:

```
it('should set items', () => {
  const result = service.setItems('Camera');
```

```
    expect(result.length).toBe(3);
});
```

Writing a test for synchronous methods, as in the previous case, is usually relatively easy; however, things are different when we want to test an asynchronous method such as the following.

This second test is a bit tricky because it involves an observable. We need to subscribe to the `getItems` method and inspect the value as soon as the observable is complete:

```
it('should get items', (done: DoneFn) => {
  service.getItems().subscribe(items => {
    expect(items.length).toBe(2);
    done();
  });
});
```

The Karma test runner does not know when an observable will complete, so we provide the `done` method to signal that the observable has been completed, and we can now assert the `expect` statement.

Testing services with dependencies

Testing services with dependencies is similar to testing components with dependencies. Every method we saw in the *Testing components* section can be applied similarly; however, we follow a different approach when testing a service that injects the `HttpClient` service.

Consider the following `deps.service.ts` file that uses the HTTP client:

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

@Injectable({
  providedIn: 'root'
})
export class DepsService {

  constructor(private http: HttpClient) { }

  getItems() {
    return this.http.get('http://some.url');
  }
}
```

```
    addItem(item: string) {
      return this.http.post('http://some.url', { name: item });
    }
}
```

Angular testing utilities provide two artifacts for mocking HTTP requests in unit tests: the `provideHttpClientTesting` function, which provides an HTTP client for testing, and the `HttpTestingController`, which mocks the `HttpClient` service. We can import both from the `@angular/common/http/testing` namespace:

```
import { TestBed } from '@angular/core/testing';
import { provideHttpClient } from '@angular/common/http';
import { HttpTestingController, provideHttpClientTesting } from '@angular/
common/http/testing';

import { DepsService } from './deps.service';

describe('DepsService', () => {
  let service: DepsService;
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        provideHttpClient(),
        provideHttpClientTesting()
      ]
    });
    service = TestBed.inject(DepsService);
    httpTestingController = TestBed.inject(HttpTestingController);
  });
});
```

Our tests should not make a real HTTP request. They only need to validate that it will be made with the correct options. The following is the first test that validates the `getItems` method:

```
it('should get items', () => {
  service.getItems().subscribe();
```

```
const req = httpTestingController.expectOne('http://some.url');
expect(req.request.method).toBe('GET');
});
```

In the preceding test, we create a fake request using the `expectOne` method of the `HttpTestingController` that takes a URL as an argument. The `expectOne` method creates a mock request object and asserts that only one request is made to the specific URL. After we have created our request, we can validate that its method is `GET`.

We follow a similar approach when testing the `addItem` method, except that we need to make sure that the body of the request contains the correct data:

```
it('should add an item', () => {
  service.addItem('Camera').subscribe();
  const req = httpTestingController.expectOne('http://some.url');
  expect(req.request.method).toBe('POST');
  expect(req.request.body).toEqual({
    name: 'Camera'
  });
});
```

After each test, we make sure that no unmatched requests are pending using the `verify` method inside an `afterEach` block:

```
afterEach(() => {
  httpTestingController.verify();
});
```

In the following section, we continue our journey through the testing world by learning how to test a pipe.

Testing pipes

As we learned in *Chapter 4, Enriching Applications Using Pipes and Directives*, a pipe is a TypeScript class that implements the `PipeTransform` interface. It exposes a `transform` method, which is usually synchronous, which means it is straightforward to test.

Consider the `list.pipe.ts` file containing a pipe that converts a comma-separated string into a list:

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

```

@Pipe({
  name: 'list'
})
export class ListPipe implements PipeTransform {

  transform(value: string): string[] {
    return value.split(',');
  }

}

```

Writing a test is simple. The only thing that we need to do is to instantiate an instance of the `ListPipe` class and verify the outcome of the `transform` method with some mock data:

```

it('should return an array', () => {
  const pipe = new ListPipe();
  expect(pipe.transform('A,B,C')).toEqual(['A', 'B', 'C']);
});

```



Angular testing utilities are not involved when testing a pipe. We create an instance of the pipe class, and we can start calling the `transform` method.

Angular directives are artifacts that we may not create very often since the built-in collection that the framework provides is more than enough; however, if we create custom directives, we should also test them. In the following section, we will learn how to accomplish this.

Testing directives

Directives are usually quite straightforward in their overall shape, being components with no view attached. The fact that directives usually work with components gives us a good idea of how to proceed when testing them.

Consider the `copyright.directive.ts` file that we created in *Chapter 5, Enriching Applications Using Pipes and Directives*:

```

import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appCopyright]'
}

```

```
)  
export class CopyrightDirective {  
  
    constructor(el: ElementRef) {  
        const currentYear = new Date().getFullYear();  
        const targetEl: HTMLElement = el.nativeElement;  
        targetEl.classList.add('copyright');  
        targetEl.textContent = `Copyright ©${currentYear} All Rights  
Reserved`;  
    }  
  
}
```

A directive is usually used with a component, so it makes sense to unit test it while using it on a component. Let's create a test host component and add it to the imports array of the testing module:

```
@Component({  
    imports: [CopyrightDirective],  
    template: '<span appCopyright></span>'  
})  
class TestHostComponent {}
```

We can now write our tests that check whether the `` element contains the `copyright` class and displays the current year in its `textContent` property:

```
describe('CopyrightDirective', () => {  
    let container: HTMLElement;  
  
    beforeEach(() => {  
        const fixture = TestBed.configureTestingModule({  
            imports: [TestHostComponent]  
        })  
        .createComponent(TestHostComponent);  
        container = fixture.nativeElement.querySelector('span');  
    });  
  
    it('should have copyright class', () => {  
        expect(container.classList).toContain('copyright');
```

```
});  
  
it('should display copyright details', () => {  
  expect(container.textContent).toContain(new Date().getFullYear().  
toString());  
});  
});
```

This is how simple it can be to test a directive. The key takeaways are that you need a component to place the directive on and that you implicitly test the directive using the component.

In the following section, we will learn how to test reactive forms.

Testing forms

As we saw in *Chapter 10, Collecting User Data with Forms*, forms are integral to an Angular application. It is rare for an Angular application not to have at least one simple form, such as a search form. In this chapter, we will focus on reactive forms because they are easier to test than template-driven forms.

Consider the following `search.component.ts` file:

```
import { Component } from '@angular/core';  
import { FormGroup, FormControl, Validators, ReactiveFormsModule } from '@  
angular/forms';  
  
@Component({  
  selector: 'app-search',  
  imports: [ReactiveFormsModule],  
  template: `  
    <form [formGroup]="searchForm" (ngSubmit)="search()">  
      <input type="text" formControlName="searchText">  
      <button type="submit" [disabled]="searchForm.invalid">Search</  
button>  
    </form>  
  `,  
})  
export class SearchComponent {  
  searchForm = new FormGroup({  
    searchText: new FormControl('', Validators.required)  
  })
```

```
});  
  
search() {  
  if(this.searchForm.valid) {  
    console.log('You searched for: ' + this.searchForm.controls.  
    searchText.value);  
  }  
}  
}
```

In the preceding component, we can write our unit tests to verify that:

- The value of the `searchText` form control can be set correctly
- The `Search` button is disabled when the form is invalid
- The `console.log` method is called when the form is valid, and the user clicks the `Search` button

To test a reactive form, we first need to import `ReactiveFormsModule` into the testing module:

```
await TestBed.configureTestingModule({  
  imports: [SearchComponent, ReactiveFormsModule]  
})  
.compileComponents();
```

For the first test, we need to assert whether the value propagates to the `searchText` form control when we type something into the input control:

```
it('should set the searchText', () => {  
  const input: HTMLInputElement = fixture.nativeElement.  
  querySelector('input');  
  input.value = 'Angular';  
  input.dispatchEvent(new CustomEvent('input'));  
  expect(component.searchForm.controls.searchText.value).toBe('Angular');  
});
```

In the preceding test, we use the `querySelector` method of the `nativeElement` property to find the `<input>` HTML element and set its value. But this alone will not be sufficient for the value to propagate to the form control. The Angular framework will not know whether the value of the `<input>` HTML element has changed until we trigger the `input` DOM event to that element. We are using the `dispatchEvent` method to trigger the event, which accepts a single method as a parameter that points to an instance of the `CustomEvent` class.

Now that we are sure that the `searchText` form control is wired up correctly, we can use it to write the remaining tests:

```
it('should disable search button', () => {
  const button: HTMLButtonElement = fixture.nativeElement.
  querySelector('button');
  component.searchForm.controls.searchText.setValue('');
  expect(button.disabled).toBeTruthy();
});

it('should log to the console', () => {
  const button: HTMLButtonElement = fixture.nativeElement.
  querySelector('button');
  const spy = spyOn(console, 'log');
  component.searchForm.controls.searchText.setValue('Angular');
  fixture.detectChanges();
  button.click();
  expect(spy).toHaveBeenCalledWith('You searched for: Angular');
});
```

Note that in the second test, we set the value of the `searchText` form control, and then we call the `detectChanges` method for the button to be enabled. Clicking on the button triggers the `submit` event of the form, and we can finally assert the expectation of our test.

In cases where a form has many controls, it is not convenient to query them inside our tests. Alternatively, we can create a `Page` object that takes care of querying HTML elements and spying on services:

```
class Page {
  get searchText() { return this.query<HTMLInputElement>('input'); }
  get submitButton() { return this.query<HTMLButtonElement>('button'); }
  private query<T>(selector: string): T {
    return fixture.nativeElement.querySelector(selector);
  }
}
```

We can then create an instance of the `Page` object in the `beforeEach` statement and access its properties and methods in our tests.

As we have seen, reactive forms are very easy to test since the form model is the single source of truth. In the following section, we will learn how to test parts of an Angular application that use the router.

Testing the router

Testing code interacting with the Angular router could easily be a separate chapter. In this section, we will focus on the following router concepts:

- Routed and routing components
- Guards
- Resolvers

Let's see first how to test routed and routing components.

Routed and routing components

A routed component is a component that is activated when we navigate to a specific application route. Consider the following `app.routes.ts` file:

```
import { Routes } from '@angular/router';
import { RoutedComponent } from './routed/routed.component';

export const routes: Routes = [
  { path: 'routed', component: RoutedComponent }
];
```

The `RoutedComponent` class is defined in the following `routed.component.ts` file:

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

@Component({
  selector: 'app-routed',
  template: '<span>{{ title }}</span>'
})
export class RoutedComponent {
  title = 'My routed component';
}
```

The preceding component binds the value of the `title` component property to a `` HTML element. The test we will write will assert if the binding works correctly.

Angular router testing is based on the component harness approach we learned about in the *Testing components* section. It exposes the `RouterTestingModule` class, which contains various utility methods for working with routed components in tests:

```
import { RouterTestingModule } from '@angular/router/testing';
```

Before we can start testing a routed component, we must register the application routing configuration in the testing module:

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    providers: [provideRouter(routes)]
  })
  .compileComponents();

  fixture = TestBed.createComponent(RoutedComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

In the preceding setup process, we provide the application routing configuration as in the `app.config.ts` file.

We have already learned that we can query the DOM of the native HTML element from the `ComponentFixture` class. When a component is loaded using the router, we use the `routeNativeElement` property instead from the `RouterTestingModule` class:

```
it('should display a span element', async () => {
  const harness = await RouterTestingModule.create();
  await harness.navigateByUrl('/routed');
  expect(harness.routeNativeElement?.querySelector('span')?.textContent).
  toBe('My routed component');
});
```

The preceding test is separated into the following steps:

1. We use the `create` method of the `RouterTestingModule` to create a new routing harness for our component.
2. We navigate to the registered route path using the `navigateByUrl` method. According to the application routing configuration, the `/routed` URL will activate the component under test.

3. We use standard query methods of the `routeNativeElement` property to verify that the `` HTML element displays the correct text.



The `RouterTestingModule` class also contains the `routeDebugElement` property, which works cross-platform similarly to the `debugElement` property of the `ComponentFixture` class.

A routing component is a component that is used to navigate to another component in an Angular application. It usually involves calling the `navigate` method of the `Router` service as follows:

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

@Component({
  selector: 'app-routed',
  template: '<span>{{ title }}</span>'
})
export class RoutedComponent {
  title = 'My routed component';

  constructor(private router: Router) {}

  goBack() {
    this.router.navigate(['/']);
  }
}
```

According to the preceding snippet, our test should verify that the router will navigate to the root path when we call the `goBack` method:

```
it('should navigate to the root path', () => {
  component.goBack();
  expect(TestBed.inject(Router).url).toBe('/');
});
```

In the preceding test, we use the `inject` method of the `TestBed` class to get a reference to the `Router` service. We then access the `url` property to verify that the navigation process was completed correctly.

In the following section, we will learn how to test router guards.

Guards

We learned in *Chapter 9, Navigating through Applications with Routing*, that router guards are plain functions.

Consider the following guard that checks the authentication status of a user:

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';

export const authGuard: CanActivateFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn) {
    return true;
  }
  return router.parseUrl('/');
};
```

In the preceding guard, we check the `isLoggedIn` property of the following `AuthService` class:

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

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  isLoggedIn = false;
}
```



We decided to keep the `AuthService` class simple and focus on the logic of the authentication guard.

If the `isLoggedIn` property is `true`, the guard also returns `true`. Otherwise, it executes the `parseUrl` method of the `Router` service to redirect users to the root path.

The Angular CLI has created the following unit test for the guard:

```
import { TestBed } from '@angular/core/testing';
import { CanActivateFn } from '@angular/router';

import { authGuard } from './auth.guard';

describe('authGuard', () => {
  const executeGuard: CanActivateFn = (...guardParameters) =>
    TestBed.runInInjectionContext(() => authGuard(...guardParameters));

  beforeEach(() => {
    TestBed.configureTestingModule({}); 
  });

  it('should be created', () => {
    expect(executeGuard).toBeTruthy();
  });
});
```

In the preceding snippet, the `executeGuard` variable encapsulates the creation of the `authGuard` function. It uses the `runInInjectionContext` method of the `TestBed` class to allow the injection of required services using the `inject` method.

To create unit tests that validate the usage of the authentication guard, we must execute the following steps:

1. Modify the `import` statement of the `@angular/router` npm package as follows:

```
import {
  ActivatedRouteSnapshot,
  CanActivateFn,
  Router,
  RouterStateSnapshot
} from '@angular/router';
```

2. Add the following `import` statement:

```
import { AuthService } from './auth.service';
```

3. Create the following variables that correspond to the injected services:

```
let authService: AuthService;
let routerSpy: jasmine.SpyObj<Router>;
```

4. Initialize the preceding variables in the `beforeEach` statement of the test suite:

```
beforeEach(() => {
  routerSpy = jasmine.createSpyObj('Router', ['parseUrl']);

  TestBed.configureTestingModule({
    providers: [
      { provide: Router, useValue: routerSpy }
    ]
  });

  authService = TestBed.inject(AuthService);
});
```

In the preceding snippet, we use the `createSpyObj` method to create a spy object for the `Router` service and provide it to the testing module. Additionally, we get the instance of the actual `AuthService` class using the `inject` method of the `TestBed` class because it is a simple service with no dependencies.

5. The first unit test should assert that the guard execution returns true when the user is authenticated:

```
it('should return true', () => {
  authService.isLoggedIn = true;
  expect(executeGuard({} as ActivatedRouteSnapshot, {} as
  RouterStateSnapshot)).toBeTrue();
});
```



We pass an empty object for the `ActivatedRouteSnapshot` and `RouterStateSnapshot` parameters because they are unnecessary in the guard.

6. The second unit test should verify that the guard execution causes a redirection to the root path:

```
it('should redirect', () => {
```

```
authService.isLoggedIn = false;
executeGuard({} as ActivatedRouteSnapshot, {} as
RouterStateSnapshot);
expect(routerSpy.parseUrl).toHaveBeenCalledWith('/');
});
```

In the following section, we will learn how to test guard resolvers.

Resolvers

Router resolvers are plain functions of a specific type similar to guards. The most common scenario when testing resolvers is to verify that the returned data is correct.

Consider the following resolver, which returns a list of items:

```
import { ResolveFn } from '@angular/router';
import { AsyncService } from './async.service';
import { inject } from '@angular/core';

export const itemsResolver: ResolveFn<string[]> = () => {
  const asyncService = inject(AsyncService);
  return asyncService.getItems();
};
```



The resolver uses the `AsyncService` we saw earlier, which returns an observable of items using the `getItems` method.

The Angular CLI will initially create the following unit test file when scaffolding the resolver:

```
import { TestBed } from '@angular/core/testing';
import { ResolveFn } from '@angular/router';

import { itemsResolver } from './items.resolver';

describe('itemsResolver', () => {
  const executeResolver: ResolveFn<boolean> = (...resolverParameters) =>
    TestBed.runInInjectionContext(() => itemsResolver(
      ...resolverParameters));

  beforeEach(() => {
```

```

    TestBed.configureTestingModule({});

});

it('should be created', () => {
  expect(executeResolver).toBeTruthy();
});
});

```

In the preceding snippet, the `executeResolver` variable encapsulates the creation of the `itemsResolver` function, similar to how it does with guards. It also uses the `runInInjectionContext` method of the `TestBed` class to allow the injection of required services.

The logic of our resolver is very simple, so we must write a single unit test:

1. Modify the `import` statement of the `@angular/router` npm package as follows:

```

import {
  ActivatedRouteSnapshot,
  ResolveFn,
  RouterStateSnapshot
} from '@angular/router';

```

2. Add the following `import` statement:

```
import { Observable } from 'rxjs';
```

3. Change the type of the `executeResolver` variable to `ResolveFn<string[]>` so that it matches the signature of the `itemsResolver` function:

```

const executeResolver: ResolveFn<string[]> = (...resolverParameters)
=>
  TestBed.runInInjectionContext(() => itemsResolver(
    ...resolverParameters));

```

4. Write the following unit test:

```

it('should return items', () => {
  (executeResolver({}) as ActivatedRouteSnapshot, {} as
  RouterStateSnapshot) as Observable<string[]>).subscribe(items => {
    expect(items).toEqual(['Microphone', 'Keyboard']);
  })
});

```

To verify that the resolver returns correct data, we must subscribe to the `executeResolver` function.

In this section, we learned how to unit test some important features of the Angular router.

Summary

We are at the end of our testing journey, and it's been a long but exciting one. In this chapter, we saw the importance of introducing unit testing in our Angular applications, the basic shape of a unit test, and the process of setting up Jasmine for our tests.

We also learned how to write robust tests for our components, directives, pipes, and services. We also discussed how to test Angular reactive forms and the router.

This unit testing chapter has almost completed the puzzle of building a complete Angular application. Only the last piece remains, which is important because web applications are ultimately destined for the web. Therefore, in the next chapter, we will learn how to produce a production build for an Angular application and deploy it to share with the rest of the world!

14

Bringing Applications to Production

A web application should typically run on the web and be accessible by anyone and from anywhere. It needs two essential ingredients: a web server hosting the application and a production build to deploy it to that server. In this chapter, we will focus on the second part of the recipe.

In a nutshell, a production build of a web application is an optimized version of the application code that is smaller, faster, and more performant. Primarily, it is a process that takes all the code files of the application, applies optimization techniques, and converts them into a single-bundle file.

In the previous chapters, we went through the many parts involved in building an Angular application. We need just one last piece to connect the dots and make our application available for anyone to use, which is to build it and deploy it to a web server.

In this chapter, we will learn about the following concepts:

- Building an Angular application
- Limiting the application bundle size
- Optimizing the application bundle
- Deploying an Angular application

Technical requirements

The chapter contains various code samples to walk you through the concept of bringing applications to production.

You can find the related source code in the ch14 folder of the following GitHub repository:

<https://www.github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Building an Angular application

To build an Angular application, we use the following command of the Angular CLI:

```
ng build
```

The build process boots up the Angular compiler, which primarily collects all TypeScript and HTML files of our application code and converts them into JavaScript. CSS stylesheet files such as SCSS are converted into pure CSS files. The build process ensures the fast and optimal rendering of our application in the browser.

An Angular application contains various TypeScript files not generally used during runtime, such as unit tests or tooling helpers. The compiler knows which files to collect for the build process by reading the `files` property of the `tsconfig.app.json` file:

```
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/app",
    "types": []
  },
  "files": [
    "src/main.ts"
  ],
  "include": [
    "src/**/*.{d,ts}"
  ]
}
```

The `src/main.ts` file is the main entry point of the application and helps Angular to go through all the components, services, and other Angular artifacts that our application needs.

The output of the `ng build` command looks like the following:

Initial chunk files size	Names	Raw size	Estimated transfer
main-N4USDVTP.js kB	main	206.91 kB	55.87
polyfills-SCHOHYNV.js kB	polyfills	34.52 kB	11.29
styles-5INURTS0.css bytes	styles	0 bytes	0
Initial total 241.44 kB			67.16
kB			

This output displays the JavaScript and CSS files generated from building the Angular application, namely:

- `main`: The actual application code that we have written
- `polyfills`: Feature polyfills for older browsers
- `styles`: Global CSS styles of our application

The Angular compiler outputs the preceding files into a `dist\appName\browser` folder, where `appName` is the application name. It also contains the following files:

- `favicon.ico`: The icon of the Angular application
- `index.html`: The main HTML file of the Angular application

The `ng build` command of the Angular CLI can be run in two modes: development and production. By default, it is run in production mode. To run it in development mode, we should run the following Angular CLI command:

```
ng build --configuration=development
```

The preceding command will have an output that looks like the following:

Initial chunk files	Names	Raw size
main.js	main	1.25 MB
polyfills.js	polyfills	90.23 kB
styles.css	styles	95 bytes
Initial total 1.35 MB		

In the preceding output, you may notice that the names of the `Initial chunk` files do not contain hash numbers, as in the case of a production build. In production mode, the Angular CLI performs various optimization techniques on the application code, such as image optimization and **Ahead of Time (AOT)** compilation, so that the final output is suitable for hosting in a web server and a production environment. The hash number added to each file ensures that the cache of a browser will quickly invalidate them upon deploying a newer version of the application.

When we ran the `ng build` command of the Angular CLI in development mode, we used the `--configuration` option. The `--configuration` option allows us to run an Angular application in different environments. We will learn how to define Angular environments in the following section.

Building for different environments

An organization may want to build an Angular application for multiple environments that require different variables, such as a backend API endpoint and application local settings. A common use case is a staging environment for testing the application before deploying it to production.

The Angular CLI enables us to define different configurations for each environment and build our application with each one. We can execute the `ng build` command while passing the configuration name as a parameter using the following syntax:

```
ng build --configuration=name
```



We can also pass a configuration in other Angular CLI commands, such as `ng serve` and `ng test`.

We can use the following Angular CLI command to start working with environments:

```
ng generate environments
```

This command will create a `src\environments` folder in the Angular project that contains the following files:

- `environment.ts`: The default environment of the application, which is used during production
- `environment.development.ts`: The application environment used during development

It will also add a `fileReplacements` section in the `angular.json` configuration file of the Angular project:

```
"development": {  
  "optimization": false,  
  "extractLicenses": false,  
  "sourceMap": true,  
  "fileReplacements": [  
    {  
      "replace": "src/environments/environment.ts",  
      "with": "src/environments/environment.development.ts"  
    }  
  ]  
}
```

In the preceding snippet, the `fileReplacements` property defines the environment file that will replace the default one while executing the `build` command in the `development` environment. If we run the `ng build --configuration=development` command, the Angular CLI will replace the `environment.ts` file with the `environment.development.ts` file in the application bundle.

Each environment file exports an `environment` object where we can define additional application properties such as the URL of a backend API:

```
export const environment = {  
  apiUrl: 'https://my-default-url'  
};
```



The same properties of the exported object must be defined in all environment files.

We need to import the default environment to access an environment property in an Angular application. For example, to use the `apiUrl` property in the main application component, we should do the following:

```
import { Component } from '@angular/core';  
import { RouterOutlet } from '@angular/router';  
import { environment } from '../environments/environment';
```

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

Not all libraries in an Angular application can be imported as a JavaScript module, as most of the Angular first-party libraries are. In the following section, we will learn how to import libraries that need the global `window` object.

Building for the window object

An Angular application may use a library like `jQuery` that must be attached to the `window` object. Other libraries, such as `Bootstrap`, have fonts, icons, and CSS files that must be included in the application bundle.

In all these cases, we need to tell the Angular CLI about their existence so that it can include them in the final bundle.

The `angular.json` configuration file contains an `options` object in the `build` configuration that we can use to define such files:

```
"options": {
  "outputPath": "dist/my-app",
  "index": "src/index.html",
  "browser": "src/main.ts",
  "polyfills": [
    "zone.js"
  ],
  "tsConfig": "tsconfig.app.json",
  "assets": [
    {
      "glob": "**/*",
      "input": "public"
    }
  ]
},
```

```
        },
      ],
      "styles": [
        "src/styles.css"
      ],
      "scripts": []
    }
}
```

The options object contains the following properties that we can use:

- `assets`: Contains static files from the `public` folder such as icons, fonts, and translations.
- `styles`: Contains external CSS stylesheet files. The global CSS stylesheet file of the application is included by default.
- `scripts`: Contains external JavaScript files.

As we add more and more features to an Angular application, the final bundle will grow bigger at some point. In the following section, we'll learn how to mitigate such an effect using budgets.

Limiting the application bundle size

As developers, we always want to build impressive applications with cool features for the end user. As such, we end up adding more and more features to our Angular application – sometimes according to the specifications and at other times to provide additional value to users. However, adding new functionality to an Angular application will cause it to grow in size, which may not be acceptable at some point. To overcome this problem, we can use **budgets**.

Budgets are thresholds that we can define in the `angular.json` configuration file, and we can make sure that the size of our application does not exceed those thresholds. To set budgets, we can use the `budgets` property of the production configuration in the build command:

```
"budgets": [
  {
    "type": "initial",
    "maximumWarning": "500kB",
    "maximumError": "1MB"
  },
  {
    "type": "anyComponentStyle",
    "maximumWarning": "4kB",
```

```
    "maximumError": "8kB"  
}  
]
```

The Angular CLI defines the preceding default budgets when creating a new Angular CLI project.

We can define a budget for different types, such as the whole Angular application or some parts of it. The threshold of a budget can be defined as bytes, kilobytes, megabytes, or a percentage of it. The Angular CLI displays a warning or throws an error when the size is reached or exceeds the defined value of the threshold.

To better understand it, let's describe the previous default example:

- A warning is shown when the size of the Angular application exceeds 500 KB and an error when it goes over 1 MB.
- A warning is shown when the size of any component style exceeds 4 KB and an error when it goes over 8.



To see all available options you can define when configuring budgets in an Angular application, check out the guide on the official documentation website at <https://angular.dev/tools/cli/build/#configuring-size-budgets>.

Budgets are great to use when we want to provide an alert mechanism in case our Angular application grows significantly. However, they are just a level of information and precaution. In the following section, we will learn how to minimize our bundle size.

Optimizing the application bundle

As we learned in the *Building an Angular application* section, the Angular CLI performs optimization techniques when we build an Angular application. The optimization process that is performed in the application code includes modern web techniques and tools, including the following:

- **Minification:** Converts multiline source files into a single line, removing white space and comments. It is a process that enables browsers to parse them faster later on.
- **Uglification:** Renames properties and methods to a non-human-readable form so that they are difficult to understand and use for malicious purposes.
- **Bundling:** Concatenates all source files of the application into a single file, called the bundle.

- **Tree-shaking:** Removes unused files and Angular artifacts, such as components and services, resulting in a smaller bundle.
- **Font optimization:** Inlines external font files in the main HTML file of the application without blocking render requests. It currently supports Google Fonts and Adobe Fonts and requires an internet connection to download them.
- **Build cache:** Caches the previous build state and restores it when we run the same build, decreasing the time taken to build the application.

If the final bundle of an Angular application remains large after all preceding optimization techniques, we can use an external tool called `source-map-explorer` to investigate the cause. Perhaps we have imported a JavaScript library twice or included an unused file. The tool analyzes our application bundle and displays all Angular artifacts and libraries we use in a visual representation. To start using it, do the following:

1. Install the `source-map-explorer` npm package from the terminal:

```
npm install source-map-explorer --save-dev
```

2. Build your Angular application and enable source maps:

```
ng build --source-map
```

3. Add the following script in the package.json file:

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build",  
  "watch": "ng build --watch --configuration development",  
  "test": "ng test",  
  "analyze": "source-map-explorer"  
}
```

4. Run the following command against the `main` bundle file:

```
npm run analyze dist/my-app/browser/main*.js
```

It will open up a visual representation of the application bundle in the browser:

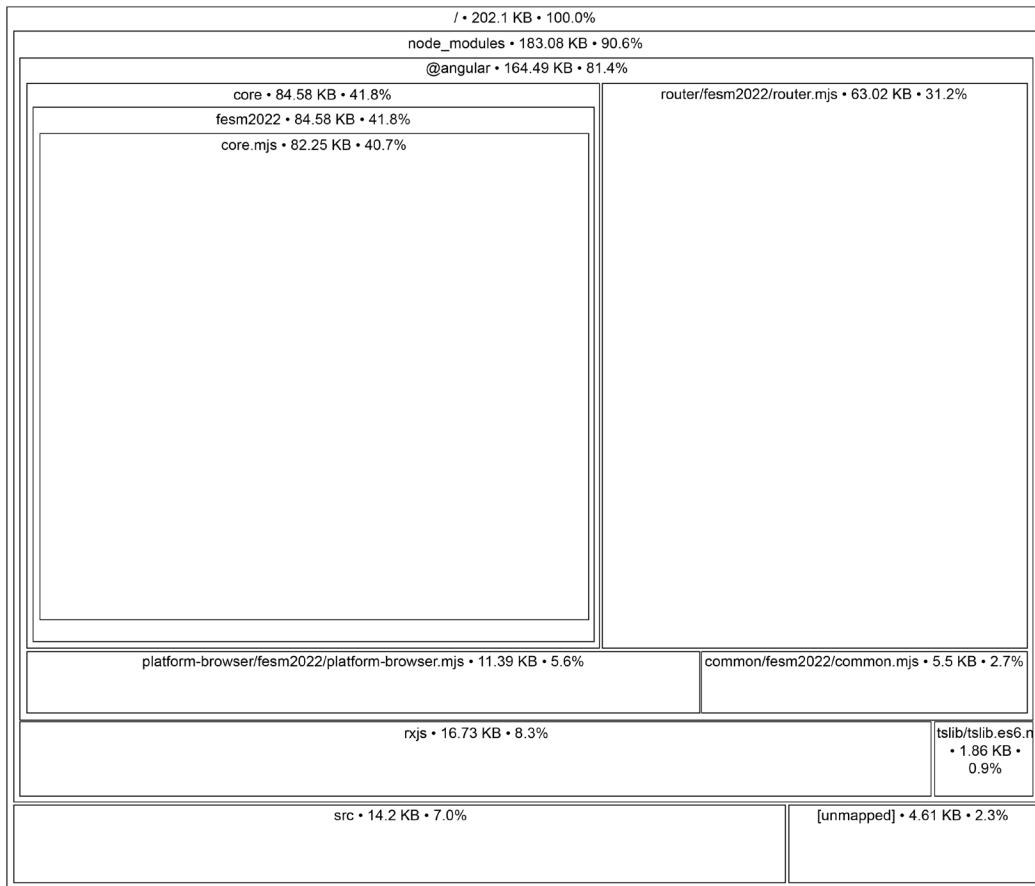


Figure 14.1: Source map explorer output

We can then interact with it and inspect it to understand why our bundle is still too large. Some causes may be the following:

- A library is included twice in the bundle
- A library that cannot be tree-shaken is included but is not currently used

The last step after we build our Angular application is to deploy it to a web server, as we will learn in the following section.

Deploying an Angular application

If you already have a web server that you want to use for your Angular application, you can copy the contents of the output folder to a path in that server. If you want to deploy it in another folder other than the root, you can change the `href` attribute of the `<base>` tag in the main HTML file in the following ways:

- Passing the `--base-href` option in the `ng build` command:

```
ng build --base-href=/mypath/
```

- Setting the `baseHref` property in the `build` command of the `angular.json` configuration file:

```
"options": {  
  "outputPath": "dist/my-app",  
  "index": "src/index.html",  
  "browser": "src/main.ts",  
  "baseHref": "/mypath/",  
  "polyfills": [  
    "zone.js"  
  ],  
  "tsConfig": "tsconfig.app.json",  
  "assets": [  
    {  
      "glob": "**/*",  
      "input": "public"  
    }  
  ],  
  "styles": [  
    "src/styles.css"  
  ],  
  "scripts": []  
}
```

If you do not want to deploy it to a custom server, you can use the Angular CLI tooling to deploy it in a supported hosting provider, which you can find at <https://angular.dev/tools/cli/deployment#automatic-deployment-with-the-cli>.

Summary

The deployment of an Angular application is the simplest and most crucial part because it finally makes your awesome application available to the end user. Web applications are all about delivering experiences to the end user at the end of the day.

In this chapter, we learned how to build an Angular application and make it ready for production. We also investigated different ways to optimize the final bundle and learned how to deploy an Angular application into a custom server, manually and automatically, for other hosting providers.

In the next chapter, which is also the final chapter of the book, we will learn how to improve the performance of an Angular application.

15

Optimizing Application Performance

As developers and technical professionals, we play a crucial role in building and deploying Angular applications, ensuring their continued performance and delivering a superior user experience. Our efforts are instrumental in the success of our applications.

The behavior of a web application and how it performs during runtime are key considerations for monitoring and optimization. We should monitor and measure application performance in case our application starts to degrade. One of the most popular metrics for identifying issues in web applications is **Core Web Vitals (CWV)**.

After determining the causes of degradation, we can apply various optimization techniques. The Angular framework provides various tools for optimizing Angular applications, including **Server-Side Rendering (SSR)**, image optimization, and deferred view loading. If we know that the application will be performance intensive beforehand, using any of the preceding tools is also highly encouraged early in development.

In this chapter, we will explore the following Angular concepts regarding optimization:

- Introducing Core Web Vitals
- Rendering SSR applications
- Optimizing image loading
- Deferring components
- Prerendering SSG applications

Technical requirements

The chapter contains various code samples to walk you through the concept of optimizing Angular applications. You can find the related source code in the ch15 folder of the following GitHub repository:

<https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition>

Introducing Core Web Vitals

CWV is a set of metrics that helps us measure the performance of a web application. It is part of **Web Vitals**, an initiative led by Google that unifies various guides and tools for measuring performance on web pages. Each metric focuses on a specific aspect of user experience, including the loading, interactivity, and visual stability of a web page:

- **Largest Contentful Paint (LCP):** This measures the load speed of a web page by calculating how long it takes for the largest element on the page to render. A fast LCP value indicates that the page becomes available to the user quickly.
- **Interaction to Next Paint (INP):** This measures the responsiveness of a web page by calculating how long it takes to respond to user interactions and provide visual feedback. A low INP value indicates that the page responds to the user quickly.
- **Cumulative Layout Shift (CLS):** This measures the stability of the UI on a web page by calculating how often unwanted layout shifts occur. A layout shift usually happens when HTML elements are moved in the DOM due to dynamic or asynchronous loading. A low CLS value indicates that the page is visually stable.



Web Vitals contains additional metrics that contribute to the existing CWV set by measuring a wider or more niche area of UX, such as **First Contentful Paint (FCP)** and **Time to First Byte (TTFB)**.

The value of each CWV metric falls into the following categories:

- **GOOD** (green)
- **NEEDS IMPROVEMENT** (orange)
- **POOR** (red)



You can find out more about CWV categories and their thresholds at <https://web.dev/articles/vitals#core-web-vitals>.

We can measure CWV in the following ways:

- **In the field:** We can use tools like **PageSpeed Insights** and **Chrome User Experience Report** while the web application runs in production.
- **Programmatically in JavaScript:** We can use standard web APIs or third-party libraries such as **web-vitals**.
- **In the lab:** We can use tools such as **Chrome DevTools** and **Lighthouse** while building the web application during development.

In this chapter, we will learn how to use Chrome DevTools to measure the performance of our e-shop application:

1. Copy the source code from *Chapter 12, Introduction to Angular Material*, into a new folder.
2. Run the following command inside the new folder to install package dependencies:

```
npm install
```

3. Run the following command to start the Angular application:

```
ng serve
```

4. Open **Google Chrome** and navigate to `http://localhost:4200`.

5. Toggle the developer tools and select the **Lighthouse** tab. Lighthouse is a tool for measuring various performance aspects of a web page, including CWV. Google Chrome has an embedded version of Lighthouse that we can use to benchmark our application:

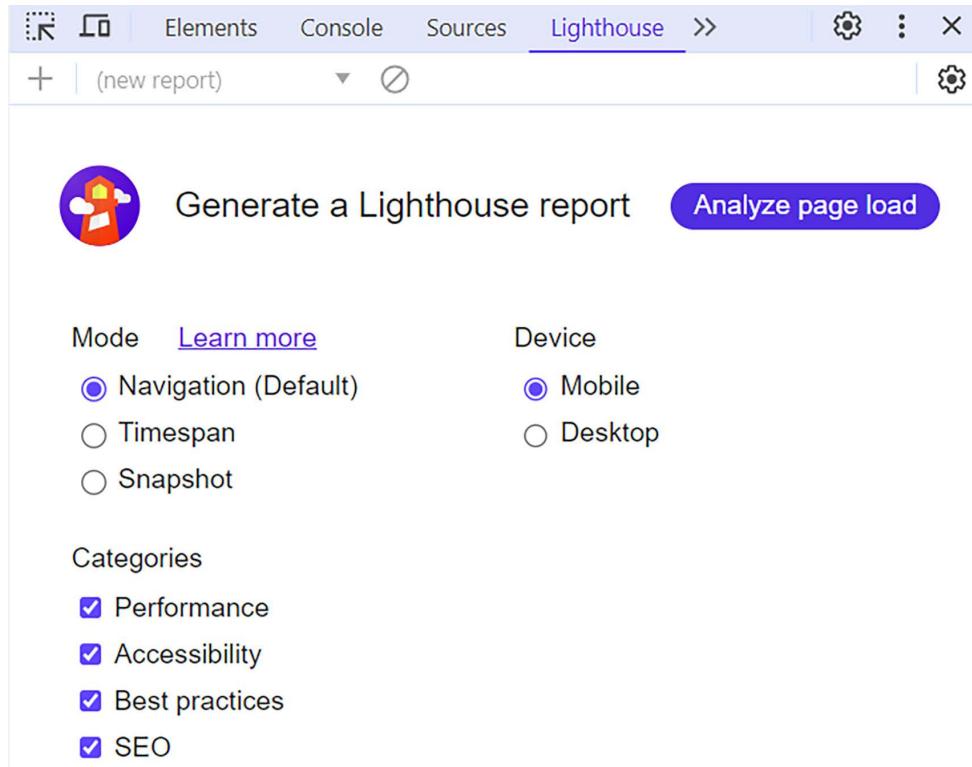


Figure 15.1: Lighthouse tab

On the screen shown in the preceding image, we can generate a Lighthouse performance report by selecting various options, including the **Device** and **Categories** sections. The **Device** section allows us to specify the environment in which we want to measure our application. The **Categories** section allows us to evaluate different metrics, including **Performance**, related to CWV.

6. Select the **Desktop** option in the **Device** section, check only the **Performance** option in the **Categories** section, and click the **Analyze page load** button:

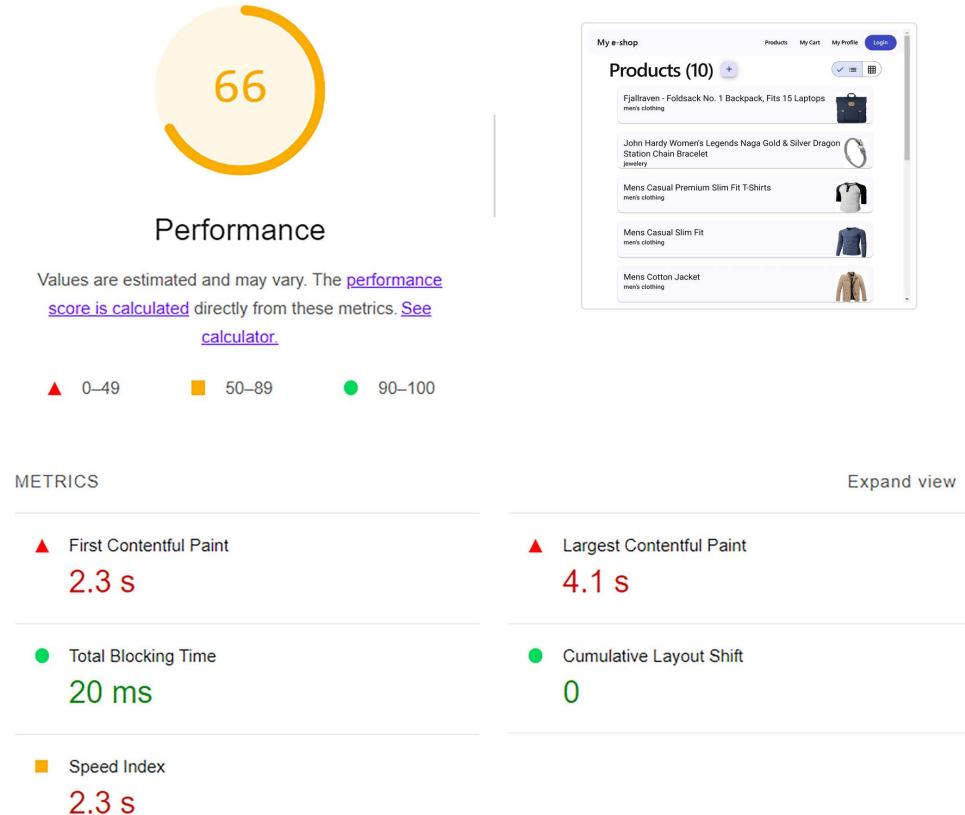


Figure 15.2: Lighthouse report

In the preceding image, we can see the individual score from CWV metrics and the overall performance score.



The overall performance score is an estimation and may vary depending on the capabilities of your computer or any installed browser extensions. It is preferable to run the benchmark in **incognito** or **private** mode to simulate an environment closer to a real-world scenario.

In the following sections, we will explore ways to improve the performance score by applying Angular best practices. We will start with SSR.

Rendering SSR applications

SSR is a technique in web development that improves application performance and security in the following ways:

- It improves the loading performance by rendering the application on the server and eliminating the initial HTML content delivered to the client. The server delivers the initial HTML to the client, which can parse and load while it waits for the JavaScript content to be downloaded.
- It improves **Search Engine Optimization (SEO)** by making the application discoverable and indexable by web crawlers. SEO provides meaningful content when shared in third-party applications such as social media platforms.
- It improves CWV metrics related to loading speed and UI stability, such as LCP, FCP, and CLS.
- It improves security by adding CSP nonces to Angular applications.

As we saw in *Chapter 1, Building Your First Angular Application*, when we created a new application using the Angular CLI, it prompted us to enable SSR:

```
Do you want to enable Server-Side Rendering (SSR) and Static Site  
Generation (SSG/Prerendering)? (y/N)
```

In our case, we have already created an Angular application using the Angular CLI. To add SSR in an existing Angular application, run the following command in a terminal window inside the Angular CLI workspace:

```
ng add @angular/ssr
```

The preceding command will ask us the following question:

```
Would you like to use the Server Routing and App Engine APIs (Developer  
Preview) for this server application? (y/N)
```

Accept the default value, No, by pressing *Enter* and the Angular CLI will prompt us to install the `@angular/ssr` npm package.



A feature in **Developer Preview** means that it is not ready yet for production but you can test it in your development environment.

After installation completes, the Angular CLI creates the following files:

- `main.server.ts`: This is used to bootstrap the application in the server using a specific configuration.
- `app.config.server.ts`: This contains the configuration for the application rendered on the server. It exports a `config` variable, which contains a merged version of the client and server application configuration files.
- `server.ts`: This configures and starts a Node.js `Express` server that renders the Angular application on the server. It uses the `CommonEngine` class from the `@angular/ssr` package to start the Angular application.

Additionally, the command will make the following modifications in the Angular CLI workspace:

- It will add the necessary options in the `build` section of the `angular.json` file to run the Angular application in SSR and SSG.
- It will add the necessary entries in the `files` and `types` property of the `tsconfig.app.json` file so that the TypeScript compiler can identify the files created for the server.
- It will add the necessary scripts and dependencies in the `package.json` file.
- It will add `provideClientHydration` in the `src\app\app.config.ts` file to enable `hydration` in the Angular application. Hydration is the process of restoring the server-side-rendered application to the client. We will learn more about hydration later in the chapter.

Now that we have installed Angular SSR in our application, let's see how to use it:

1. Open the `app.config.ts` file and modify the `import` statement of the `@angular/common/http` namespace as follows:

```
import { provideHttpClient, withFetch } from '@angular/common/http';
```

The `withFetch` method is used to configure the Angular HTTP client so that it uses the native `fetch` API for making requests.

It's strongly recommended to enable `fetch` for applications that use SSR for better performance and compatibility.

2. Pass the `withFetch` method as a parameter in the `provideHttpClient` method:

```
provideHttpClient(withFetch())
```

3. Run the following command to build the Angular application:

```
ng build
```

The preceding command generates browser and server bundles inside the `dist\my-app` folder and prerenders static routes. We will learn more about prerendering in the *Prerendering SSG applications* section.

- Run the following command to run the SSR application:

```
npm run serve:ssr:my-app
```

The preceding command will start the Express server locally at port 4000 and serve the SSR application.

- Open Google Chrome and navigate to `http://localhost:4000`. You should see the e-shop application on the web page.
- Repeat the process we learned in the previous section to run a performance benchmark using Lighthouse. The overall score and CWV metrics should have been improved dramatically:

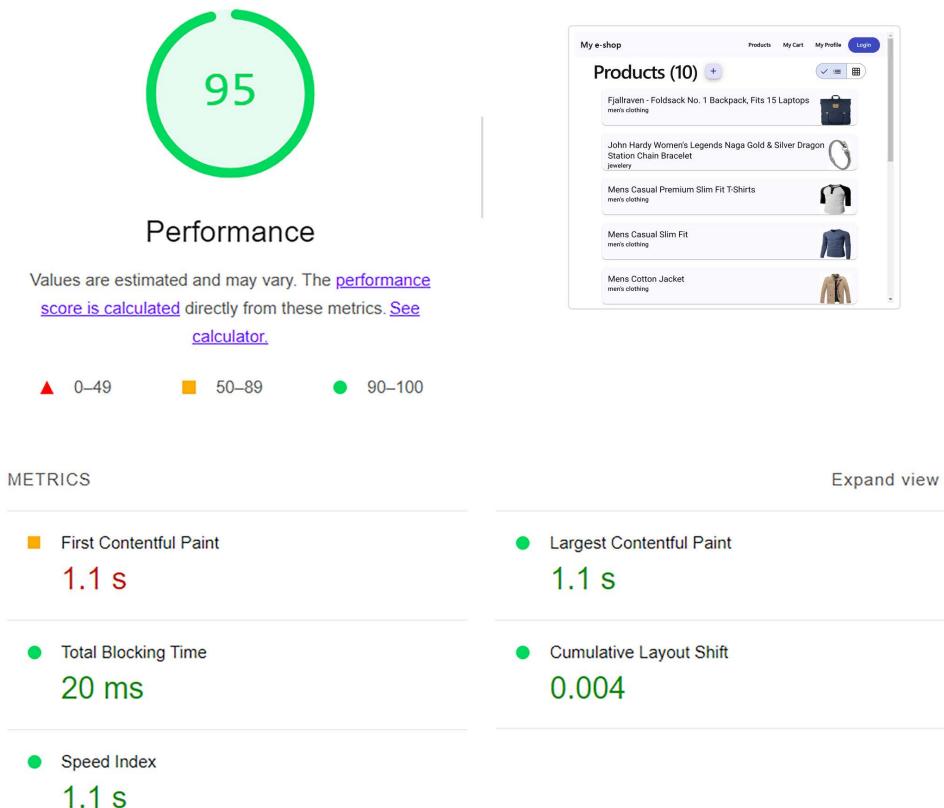


Figure 15.3: Lighthouse report (SSR)

The performance of our application has been improved by more than 20% just by installing SSR in our Angular application! As we will learn later in the chapter, we can apply various Angular techniques to improve performance further more.

Angular SSR is a good fit when we need to fetch data from the server and display it statically on a website. However, there are cases where SSR is not beneficial, such as when an application is based on data entry and has a lot of user inputs.

In the following section, we will learn how to override SSR or skip it completely for certain parts of an Angular application.

Overriding SSR in Angular applications

Hydration is an important feature enabled by default in Angular SSR applications. It improves the overall performance of the application by handling the creation of the DOM on the client efficiently. The client can reuse the DOM structure of the server-side-rendered application instead of creating it from scratch and forcing a UI flicker, which affects CWV metrics such as LCP and CLS. The hydration process will fail in the following cases:

- When we try to manipulate the DOM through a native browser API such as `window` or `document` either directly or using a third-party library
- When our component templates do not have a valid HTML syntax

We can overcome the preceding problems by applying the following best practices:

- Use Angular APIs to detect the platform on which our application is running before interacting with the DOM
- Skip hydration for specific Angular components

Let's see how to use both with an example:

1. Run the SSR version of the Angular application, as shown in the previous section.
2. Notice the text that is displayed in the application footer:

- v1.0

The copyright information is not displayed correctly.

3. Open the `copyright.directive.ts` file and focus on the constructor code:

```
constructor(el: ElementRef) {
  const currentYear = new Date().getFullYear();
  const targetEl: HTMLElement = el.nativeElement;
```

```
targetEl.classList.add('copyright');
targetEl.textContent = `Copyright ©${currentYear} All Rights
Reserved`;
}
```

The preceding code uses the `nativeElement` property to manipulate the DOM by adding a CSS class and setting `textContent` of the HTML element. However, as mentioned, the code breaks our application because there is no DOM on the server. Let's fix that!

4. Open the `app.component.html` file and add the `ngSkipHydration` attribute on the `<mat-toolbar>` element of the `<footer>` HTML tag:

```
<footer>
  <mat-toolbar ngSkipHydration>
    <mat-toolbar-row>
      <span appCopyright> - v{{ settings.version }}</span>
    </mat-toolbar-row>
  </mat-toolbar>
</footer>
```



`ngSkipHydration` is an HTML attribute, not an Angular directive. It can only be used in other Angular components, not native HTML elements. It would not work if we had added it in the `<footer>` tag instead.

In the preceding snippet, the `<mat-toolbar>` component and its child components will not be hydrated. This effectively means that Angular will create them from scratch when the SSR version of the application is ready.

5. Run step 1 again and observe the output in the application footer:

Copyright ©2024 All Rights Reserved - v1.0



Skipping hydration should be considered a workaround. We use it temporarily in cases where hydration cannot be enabled. It is recommended to refactor your code so your application can benefit from hydration capabilities.

An alternate and better approach is to refactor our code so that it executes client code conditionally:

1. Modify the `import` statements in the `copyright.directive.ts` file as follows:

```
import { isPlatformBrowser } from '@angular/common';
import { Directive, ElementRef, inject, OnInit, PLATFORM_ID } from
  '@angular/core';
```

The `PLATFORM_ID` is an `InjectionToken` that indicates the type of platform our application is currently running on. The `isPlatformBrowser` function checks if a given platform ID is the browser.

Add the `OnInit` interface to the list of implemented interfaces of the `CopyrightDirective` class:

```
export class CopyrightDirective implements OnInit
```

2. Add the following class properties:

```
private platform = inject(PLATFORM_ID);
private el = inject(ElementRef);
```

3. Remove the constructor and add the following `ngOnInit` method:

```
ngOnInit(): void {
  if (isPlatformBrowser(this.platform)) {
    const currentYear = new Date().getFullYear();
    const targetEl: HTMLElement = this.el.nativeElement;
    targetEl.classList.add('copyright');
    targetEl.textContent = `Copyright ©${currentYear} All Rights
  Reserved ${targetEl.textContent}`;
  }
}
```

The `isPlatformBrowser` function accepts the platform ID as a parameter.



Angular also provides the `isPlatformServer` function, a counterpart of the `isPlatformBrowser` function, which checks if the current platform is the server.

4. Build and run the application in server-side mode to verify that the copyright message is still visible.

To sum up, it is recommended that you use Angular SSR throughout your application and refactor parts of the application code that must run on the browser. This will allow you to reap all the benefits of a server-side-rendered application.

In the preceding section, we showed that adding SSR to an Angular application dramatically improves its overall performance score. As we will learn in the following section, we can do even better by applying optimization techniques to product images.

Optimizing image loading

The product list, which is the landing component of our application, displays an image of each product on the list. How images are loaded in an Angular application can affect CWV metrics such as LCP and CLS. Our application currently loads images as received from the Fake Store API. However, we can use specific Angular artifacts to enforce best practices while loading images.

The Angular framework provides us with the `NgOptimizedImage` directive, which we can attach to `` HTML elements:

1. Open the `product-list.component.ts` file and import the `NgOptimizedImage` class from the `@angular/common` npm package:

```
import { AsyncPipe, CurrencyPipe, NgOptimizedImage } from '@angular/common';
```

2. Add the `NgOptimizedImage` class in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    AsyncPipe,
    CurrencyPipe,
    RouterLink,
    MatMiniFabButton,
    MatIcon,
    MatCardModule,
    MatTableModule,
    MatButtonToggle,
    MatButtonToggleGroup,
    NgOptimizedImage
  ],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

3. Open the `product-list.component.html` file and replace the binding of the `src` property with the `ngSrc` directive:

```
<mat-card-title-group>
  <mat-card-title>{{ product.title }}</mat-card-title>
  <mat-card-subtitle>{{ product.category }}</mat-card-subtitle>
  <img mat-card-sm-image [ngSrc]="product.image" />
</mat-card-title-group>
```

The `ngSrc` directive is insufficient to prevent layout shifts while loading the image. We must also set the image size by defining the `width`, `height`, or `fill` attributes. In this case, we will use the latter because the size of each image is not the same for all products:

```
<img mat-card-sm-image [ngSrc]="product.image" fill />
```

4. Open the `product-list.component.css` file and add the following CSS styles to position the image at the top right of the container:

```
img {
  object-fit: contain;
  object-position: right 5px top 0;
}
```

5. Run the following command to start the application:

```
ng serve
```

6. Navigate to `http://localhost:4200` and verify that the product list is shown correctly.

The benefits acquired from using the `NgOptimizedImage` directive are not noticeable in the UI at once. The directive works in the background and automatically improves the LCP metric of CWV by:

- Setting fetch priority on the `` HTML element
- Lazy loading images
- Setting preconnect link tags and preload hints in the case of SSR
- Generating `srcset` attributes for responsive images

Additionally, it helps developers to follow best practices regarding image loading, such as:

- Setting the size of the image if it is known beforehand
- Loading images through a CDN
- Displaying appropriate warnings in the console window for different metrics

The `NgOptimizedImage` directive contains many other features we can enable to achieve powerful performance improvements, such as setting up image loaders, using placeholders, and defining priority images to load. You can find more information at <https://angular.dev/guide/image-optimization>.

We have already learned about various tools for improving application performance. One of the most performant tools is **deferrable views**, which we will learn about in the following section.

Deferring components

Introducing the new control flow syntax enabled Angular to integrate new primitives in the framework, improving the ergonomics, DX, and performance of Angular applications. One such primitive is deferrable views, which allow lazily loading an Angular component and its dependencies.

Introducing deferrable views

We have already learned how to use the Angular router for lazy loading a component based on a specific route. Deferrable views provide a new API that supplements the preceding one. Combining it with lazy-load routing guarantees the development of high-performance and powerful web applications. Deferrable views allow us to lazy load a component based on an event or the component state and have the following characteristics:

- They are simple to use and easy to reason about the enclosed code
- We define them in a declarative way
- They minimize the initial application load and final bundle size, improving CWV metrics such as LCP and TTFB

Each deferrable view is split into a separate chunk, similar to the individual chunk files generated by lazy-loaded routes. They consist of the following HTML blocks:

- `@defer`: Indicates the HTML content that will be loaded.
- `@placeholder`: Indicates the HTML content shown before the `@defer` block starts loading. It is particularly useful when the application is loaded over a slow network or when we want to avoid UI flickering.
- `@loading`: Indicates the HTML content that will be visible while the `@defer` block is loading.
- `@error`: Indicates the HTML content shown if an error occurs while the `@defer` block is loading.

We will learn how to use each block in the following section.

Using deferrable blocks

We will integrate deferrable views in our e-shop application by creating a component that displays a featured product from the Fake Store API that is not currently in the product list. Let's start:

1. Run the following command to create the new component:

```
ng generate component featured
```

2. Open the `products.service.ts` file and add the following method, which gets a specific product with ID 20 from the Fake Store API:

```
getFeatured(): Observable<Product> {
  return this.http.get<Product>(this.productsUrl + '/20');
}
```

3. Open the `featured.component.ts` file and modify the import statements as follows:

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MatButton } from '@angular/material/button';
import { MatCardModule } from '@angular/material/card';
import { Observable } from 'rxjs';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

4. Modify the `imports` array of the `@Component` decorator as follows:

```
@Component({
  selector: 'app-featured',
  imports: [CommonModule, MatButton, MatCardModule],
  templateUrl: './featured.component.html',
  styleUrls: ['./featured.component.css']
})
```

5. Modify the `FeaturedComponent` class as follows:

```
export class FeaturedComponent implements OnInit {
  product$: Observable<Product> | undefined;

  constructor(private productService: ProductsService) {}

  ngOnInit() {
```

```
    this.product$ = this.productService.getFeatured();
}
}
```

In the preceding TypeScript class, we have declared the product\$ observable and assigned it to the returned value of the getFeatured method from the ProductsService class.

6. Open the featured.component.html file and replace its contents with the following HTML code:

```
@if (product$ | async; as product) {
  <mat-card>
    <mat-card-header>
      <mat-card-title>MEGA DEAL</mat-card-title>
      <mat-card-subtitle>{{ product.title }}</mat-card-subtitle>
    </mat-card-header>
    <img mat-card-image [src]="product.image" />
    <mat-card-actions>
      <button mat-flat-button color="primary">Buy now</button>
    </mat-card-actions>
  </mat-card>
}
```

In the preceding snippet, we use the `async` pipe to subscribe to the `product$` observable inside the `@if` block. The HTML content of the block displays product details as an Angular Material card component.

7. Open the featured.component.css file and add the following CSS styles for the card and the button components:

```
mat-card {
  max-width: 350px;
}

button {
  width: 100%;
}
```

The new Angular component is in place. We must add it to the main component of the application and use a `@defer` block to load it:

1. Open the `app.component.ts` file and add the following `import` statement:

```
import { FeaturedComponent } from './featured/featured.component';
```

2. Add the `FeaturedComponent` class in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-root',
  imports: [
    RouterOutlet,
    RouterLink,
    CopyrightDirective,
    AuthComponent,
    MatToolbarRow,
    MatToolbar,
    MatButton,
    MatBadge,
    FeaturedComponent
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

3. Open the `app.component.html` file and add the `<app-featured>` component inside the `<main>` HTML tag:

```
<main class="main">
  <div class="content">
    <router-outlet />
  </div>
  @defer() {
    <app-featured />
  }
</main>
```

In the preceding snippet, we use the `@defer` block to declare the `<app-featured>` component using the self-enclosing tag syntax.

- Run the `ng serve` command to start the application and observe the **Lazy chunk files** section in the terminal window:

Lazy chunk files	Names	Raw size
chunk-0P24QI45.mjs	featured-component	2.88 kB
chunk-4T4L5V7V.mjs	user-routes	1.19 kB

The source code of the featured component is split into a chunk file.

- Navigate to `http://localhost:4200` and observe the new component on the right side of the product list:



Figure 15.4: Featured product

Try to reload the browser, and you will notice a UI flickering while loading the featured product. We will use the `@placeholder` block to display an outline image before the featured component starts loading:

1. Copy the `placeholder.png` image from the `public` folder of the GitHub repository described in the *Technical requirements* section to the respective folder of your workspace.
2. Add a `@placeholder` block following the `@defer` block as follows:

```
@defer() {  
  <app-featured />  
} @placeholder(minimum 1s) {  
    
}
```

The `@placeholder` block accepts an optional parameter defining the `minimum` time the placeholder will be visible. In this case, we have defined the minimum time as 1 second.

3. Run the application using the `ng serve` command and verify that the following placeholder image is visible for 1 second before the actual content is loaded:

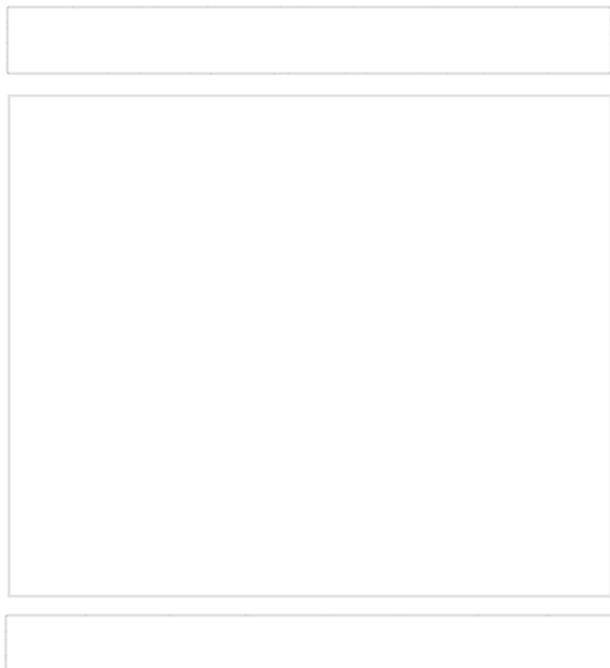


Figure 15.5: Placeholder image

An alternate approach would be to use the `@loading` block and display a loading indicator, such as a spinner, while the featured component is loading:

1. Open the `app.component.ts` file and add the following `import` statement:

```
import { MatProgressSpinner } from '@angular/material/progress-
spinner';
```

The `MatProgressSpinner` class is a spinner component from the Angular Material library.

2. Add the `MatProgressSpinner` class in the `imports` array of the `@Component` decorator:

```
@Component({
  selector: 'app-root',
  imports: [
    RouterOutlet,
    RouterLink,
    CopyrightDirective,
    AuthComponent,
    MatToolbarRow,
    MatToolbar,
    MatButton,
    MatBadge,
    FeaturedComponent,
    MatProgressSpinner
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

3. Add the `@loading` block in the `app.component.html` file as follows:

```
@defer() {
  <app-featured />
} @loading(minimum 1s) {
  <mat-spinner ngSkipHydration></mat-spinner>
}
```

The `@loading` block accepts the same optional parameters as the `@placeholder` block. In this case, we show the spinner component for 1 second minimum.



We added the `ngSkipHydration` attribute because the spinner component interacts with the browser DOM and cannot be hydrated.

4. If we run the application using the `ng serve` command, we should see a spinner indication for 1 second while the featured component is loading.

The `@error` block in deferrable views works similarly to the `@placeholder` and `@loading` blocks. The HTML content inside it will be visible when an error occurs while loading the `@defer` block contents:

```
@defer() {
  <app-featured />
} @placeholder(minimum 1s) {
  
} @error() {
  <span>An error occurred while loading the featured product</span>
}
```

As we have seen, the contents of a `@defer` block start loading immediately when the component that it belongs to is rendered. However, the deferrable views API provides us with ergonomic tools to control when the block will be loaded, as we will see in the following section.

Loading patterns in `@defer` blocks

Using `triggers` and `prefetch` mechanisms, we can control how and when a `@defer` block will load:

- Triggers define when the block's contents start loading
- Prefetch defines whether Angular will fetch the contents beforehand so that they are available when needed

We can define a trigger as an optional parameter inside the `@defer` block using the `on` keyword and the name of the trigger:

```
@defer(on viewport) {
  <app-featured />
} @placeholder(minimum 1s) {
  
} @error() {
```

```
<span>An error occurred while loading the featured product</span>
}
```

The Angular framework contains the following built-in triggers:

- **viewport**: This will trigger the block when the content enters the browser **viewport**, which is the part of the browser that is currently visible.



You can learn more about the viewport at <https://developer.mozilla.org/docs/Glossary/Viewport>.

- **interaction**: This will trigger the block when the user interacts with the content.
- **hover**: This will trigger the block when users hover over the area covered by the content with their mouse.
- **idle**: This will trigger the block when the browser has entered an **idle** state, which is the default behavior of deferrable views. The idle state of the browser is triggered by the native `requestIdleCallback` API.



You can learn more about the idle state at <https://developer.mozilla.org/docs/Web/API/Window/requestIdleCallback>.

- **immediate**: This will trigger the block when the client renders the page.



The difference between not using the block and using it with the **immediate** trigger is that we benefit from the code-splitting features of deferrable views and deliver less JavaScript to the client.

- **timer**: This will trigger the block after a specified duration. The duration is a required parameter of the **timer** function:

```
@defer(on timer(2s)) {
  <app-featured />
}
```

- The preceding snippet will start loading the featured component after 2 seconds.

We can achieve better loading granularity by combining triggers:

```
@defer(on timer(2s); on idle) {  
  <app-featured />  
}
```

The preceding snippet will load the featured component when the browser is `idle` or after 2 seconds.

In addition to the built-in triggers, we can create custom triggers by ourselves using the `when` keyword. The `when` keyword is followed by an expression that evaluates to a boolean:

```
@defer(when isActive === true) {  
  <app-featured />  
}
```

In the preceding snippet, the featured component will be loaded when the `isActive` component property is `true`.

Triggers in deferrable views are powerful and ergonomic tools that can give amazing results in speed and performance. When combined with prefetching, they can achieve great performance improvements in Angular applications. Prefetching allows us to specify the condition in which we can prefetch a deferrable view to be ready when needed. Prefetching supports all built-in triggers of deferrable views:

```
@defer(on timer(2s); prefetch on idle) {  
  <app-featured />  
}
```

The preceding snippet will prefetch the content when the browser is `idle` and load it after 2 seconds. It can also define when it will prefetch the content using the `when` keyword or create custom triggers.

Triggers and prefetching allow us to create sophisticated and complex scenarios for loading deferrable views. The versatility that the deferrable views API provides makes it a very useful tool in developing highly sophisticated and performant Angular applications.



Deferrable views should not be used for content that must be rendered immediately.

In the following section, we will conclude our journey to optimizing application performance with Angular SSG.

Prerendering SSG applications

SSG or build-time prerendering is the process of creating static generated HTML files for an Angular application. It happens by default when we build an Angular SSR application using the `ng build` Angular CLI command.

The main benefit of an SSG application is that it does not require round-trip times between the server and client for each request. Instead, every page is served as static content, eliminating the time it takes to load the application, as measured by the TTFB CWV metric.

In the *Rendering SSR applications* section, the output of the Angular CLI build command included the following message:

```
Prerendered 4 static routes.
```

Let's see how SSG works and what the preceding output means:

1. Run the following command to build the Angular application:

```
ng build
```

2. The `ng build` command will create the `dist\my-app\browser` folder.



The preceding folder should not be confused with the `browser` folder generated when building a non-SSR Angular application.

3. Navigate to the `dist\my-app` folder and open the `prerendered-routes.json` file:

```
{
  "routes": [
    "/cart",
    "/products",
    "/products/new",
    "/user"
  ]
}
```

It lists the application routes that Angular SSG prerendered. It has also created one folder and `index.html` file for each route inside the `browser` folder.

4. Open the `products\index.html` file, and you will see that Angular has added all CSS and HTML files, and it has even rendered the product data as fetched from the Fake Store API.
5. To preview how SSG works, run the `ng serve` command to start the application and navigate to `http://localhost:4200/products`. The product list loads instantly without waiting for the application to fetch data from the Fake Store API.



The `ng serve` command serves the SSG version of our application because it executes the `ng build` command under the hood. To disable SSG, open the `angular.json` file and set the `prerender` property to `false` inside the `build` section.

SSG is enabled by default in Angular SSR applications and can dramatically improve their loading time and runtime performance. It can be particularly useful for low-end devices with poor performance.

Summary

In this chapter, we learned different ways to optimize and improve the performance of an Angular application. We introduced the concept of CWV and how it can affect a web application. We explored how to measure and improve CWV metrics using SSR and hydration in Angular applications. We also investigated different aspects of performance optimizations, such as the `NgOptimizedImage` directive and deferrable views. Finally, we saw an overview of SSG in Angular applications.

Our journey with the Angular framework ends with this chapter. However, the possibilities of what we can do are endless. The Angular framework is updated with new features in each release, giving web developers a powerful tool for everyday development. We were delighted to have you on board, and we hope this book has helped you to broaden your ideas on what you can achieve with such an excellent tool!

Join us on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/LearningAngular5e>





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

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.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

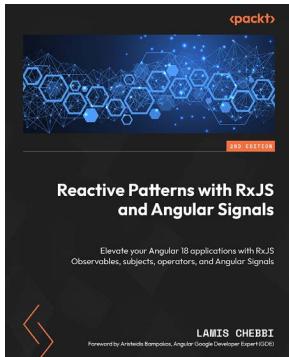


Effective Angular

Roberto Heckers

ISBN: 978-1-80512-553-2

- Create Nx monorepos ready to handle hundreds of Angular applications
- Reduce complexity in Angular with the standalone API, inject function, control flow, and Signals
- Effectively manage application state using Signals, RxJS, and NgRx
- Build dynamic components with projection, TemplateRef, and defer blocks
- Perform end-to-end and unit testing in Angular with Cypress and Jest
- Optimize Angular performance, prevent bad practices, and automate deployments

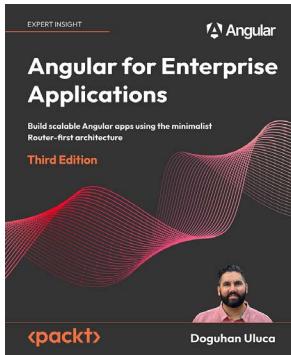


Reactive Patterns with RxJS and Angular Signals

Lamis Chebbi

ISBN: 978-1-83508-770-1

- Get to grips with RxJS core concepts such as Observables, subjects, and operators
- Use the marble diagram in reactive patterns
- Delve into stream manipulation, including transforming and combining them
- Understand memory leak problems using RxJS and best practices to avoid them
- Build reactive patterns using Angular Signals and RxJS
- Explore different testing strategies for RxJS apps
- Discover multicasting in RxJS and how it can resolve complex problems
- Build a complete Angular app reactively using the latest features of RxJS and Angular



Angular for Enterprise Applications, Third Edition

Doguhan Uluca

ISBN: 978-1-80512-712-3

- Best practices for architecting and leading enterprise projects
- Minimalist, value-first approach to delivering web apps
- How standalone components, services, providers, modules, lazy loading, and directives work in Angular
- Manage your app's data reactivity using Signals or RxJS
- State management for your Angular apps with NgRx
- Angular ecosystem to build and deliver enterprise applications
- Automated testing and CI/CD to deliver high quality apps
- Authentication and authorization
- Building role-based access control with REST and GraphQL

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.

Share your thoughts

Now you've finished *Learning Angular, Fifth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

401 Unauthorized error

responding to 315, 316

@Host decorators

reference link 146

@Self decorators

reference link 146

A

Ahead of Time (AOT) compilation 408

Angular application

bootstrapping 13, 14

building 406-408

building, for different environments 408-410

building, for window object 410, 411

components 13

deploying 415

structure 12

template syntax 14-16

Angular CDK 321

Angular characteristics

cross-platform 4

onboarding 5

tooling 4, 5

worldwide, usage 5, 6

Angular CLI 1

commands 8, 9

installing 7, 8

new project, creating 9-11

prerequisites 6

workspace, setting up 6

Angular component 13

creating 56

creating, with Angular CLI 58-60

inter-communication 75

structure 56-58

Angular components, inter-communication

data, emitting through custom events 80, 81

data, passing with input binding 75-77

events, listening with output binding 77-80

template reference variables 81, 82

Angular decorator 57

Angular Dependency Injection (DI) 121-123

dependencies 129-132

Angular DevTools 2, 16-20

Angular directive 65

Angular form

reacting, to state changes 304, 305

state, manipulating 303

state, updating 303

Angular framework

online communities 5

- Angular HTTP client** 189-191
 - data, fetching 192-201
 - data, modifying 202
 - used, for communicating data services 188, 189
 - used, for handling CRUD data 192
- Angular HTTP client authentication** 214
 - with backend API 214, 215
- Angular HTTP client authorization** 214
 - HTTP requests 218-221
 - user access 216-218
- Angular HTTP client, data modification**
 - new product, adding 203-207
 - product price, updating 207-210
 - product, removing 210-213
- AngularJS** 3
- Angular Language Service** 22, 23
- Angular Material** 320, 321
 - installing 321-324
 - UI components, adding 324, 325
 - UI components, integrating 329
 - UI components, theming 325-329
- Angular router** 224, 225
 - base path, specifying 226
 - components, rendering 228
 - configuring 227, 228
 - enabling, in Angular applications 226
 - main routes, configuring 228-232
- Angular service**
 - creating 124, 125
 - injecting, in constructor 126-128
 - inject keyword 128, 129
- Angular testing utilities** 365
- any type** 43
- application bundle**
 - optimizing 412-414
 - size, limiting 411, 412
- application routes**
 - built-in route paths, using 238, 239
 - navigating, imperatively to 233-237
 - organizing 232, 233
 - router links, styling 239, 240
- Array.push method** 112
- Array.slice method** 147
- array type** 42
- arrow functions** 32, 33
- assertion** 364
- asynchronous information**
 - callback hell, shifting to promises 156-160
 - handling strategies 156
 - observables 160-162
- asynchronous methods**
 - testing 386
- asynchronous services**
 - components, testing 378-380
- async pipe**
 - using 174, 175
- B**
 - badge component** 355
 - badges**
 - applying 355, 356
 - boolean type** 42
 - Bootstrap** 410
 - bootstrapping** 13
 - budgets** 411
 - built-in route paths**
 - using 238, 239
- C**
 - callback hell** 158
 - callback pattern** 156

- change detection strategy**
 - deciding on 85-89
- child routes**
 - used, for reusing components 245-247
- Chrome DevTools 419**
 - used, for measuring performance of e-shop application 419, 420
- Chrome User Experience Report 419**
- class provider syntax 147**
- class statement**
 - elements 35
- component harness 383, 384**
- component lifecycle 89, 90**
 - child components, accessing 95, 96
 - initialization, performing 90, 91
 - input binding changes, detecting 93-95
 - resources, cleaning up 91-93
- components**
 - deferring 430
 - testing 366-370
 - testing, with asynchronous services 378-380
 - testing, with component harness 383-385
 - testing, with dependencies 370
 - testing, with inputs and outputs 380-383
- component template**
 - class binding 71, 72
 - data, displaying conditionally 63-66
 - data, displaying from component class 62, 63
 - data, obtaining from 73, 74
 - data representation, controlling 63
 - interacting with 60
 - iterating, through data 66-69
 - loading 60, 61
 - style binding 72
- styling 71**
 - switching, through 69-71
- computed signals 113, 178-180**
 - working 180, 181
- constructor injection pattern 127**
- Core Web Vitals (CWVs) 417, 418**
 - Cumulative Layout Shift (CLS) 418
 - Interaction to Next Paint (INP) 418
 - Largest Contentful Paint (LCP) 418
 - measuring, ways 419
 - reference link 419
- Create Read Update Delete (CRUD) 191**
 - data handling, in Angular HTTP client 192
- CSS classes**
 - global validation with 288-290
- CSS styling**
 - encapsulating 82-85
- custom types 43, 44**
- custom validators**
 - building 297-302
- D**
- data**
 - manipulating, with pipes 99
 - sorting, with pipes 106-109
- data services**
 - communicating, with Angular HTTP client 188, 189
- default route 238**
- deferrable blocks**
 - using 431-437
- deferrable views 263, 430**
- DELETE operation 189**
- dependencies 122**
 - components, testing 370

replacing, with stub 371-375
services, testing 387-389

dependency method

spying on 375-378

developer experience (DX) 3

dialog component

configuring 353, 354
confirmation dialog, creating 350-353
data, obtaining from 354, 355
user notifications, displaying 355

directives 99

attribute directives 114
building 113, 114
components 113
dynamic data, displaying 114-118
property binding 118-120
structural directives 113
testing 390-392

E

EditorConfig 24, 25
element injectors 132
ElementRef 116
emit method 80
end-to-end (E2E) testing 368
environment injectors 132
event binding 73

F

Fake Store API 191
reference link 191
fallback route 228
fetch API
reference link 188
filter method 162, 167

First Contentful Paint (FCP) 418

forms, in Angular applications

custom validators, building 297-302
global validation, with CSS 288-290
input, validating 288
testing 392-394
validation, in reactive forms 294-297
validation, in template-driven forms 290-294

framework errors

demystifying 316-318

function parameters 31, 32

G

generics 50, 51
GET operation 189
Git repository 7
global error handler
creating 312-315

H

HTTP backend API
setting up 191, 192
HttpClientModule 190
HTTP headers 218
HttpParams class 196
HTTP request errors
catching 308-312
hydration 423-425

I

image loading
optimizing 428-430
inject function
reference link 129

injector 122

injector hierarchy

- objects, transforming in
 - Angular services 151-153
- service implementation, overriding 147-149
- services, providing 149-151
- used, for overriding providers 147

injector services 133

- dependencies, sharing through
 - components 133-137
- provider lookup, restricting 145, 146
- root and component injectors 138, 139
- sandboxing components, with multiple instances 139-144

input binding

- used, for passing data 75-77

Integrated Development Environment (IDE) 2

Interaction to Next Paint (INP) 418

interceptors 219

interfaces 48

interpolation 15

J

Jasmine 363-365

JavaScript features 28

- arrow functions 32, 33
- classes 35, 36
- function parameters 31, 32
- modules 36, 37
- nullish coalescing 34
- optional chaining 33, 34
- variable declaration 28-30

jQuery 410

K

Karma 365

key logger

- implementing 162-165

keyUp event feature 162

L

Largest Contentful Paint (LCP) 418

lazy-loaded route

- protecting 262, 263

lazy loading 259

Lighthouse 419

Long Time Support (LTS) 6

M

main routes

- configuring 228-232

map method 162-167

map operator 165

matcher function 364

Material Design 320

Material Icon Theme 24

monkey patch 85

N

navigation, with advanced features

- enhancing 252

lazy loading 259-262

preventing, away from route 254, 255

route access, controlling 252-254

route data, prefetching 256-259

Node.js 6

npm 7

nullish coalescing 34

number type 42

O

observables

- creating 166
- subscribing 169-172
- transforming 167-169

observable streams 155

observables, unsubscribing 172

- async pipe, using 174, 175
- component, destroying 172-174

observer methods

- complete 189
- error 189
- next 189

observer pattern 160

one-way binding 267

onKeyPress method 119

operators 163

optional chaining 33, 34

output binding

- used, for listening for events 77-80

P

PageSpeed Insights 419

patterns

- loading, in @defer blocks 437-439

pipes 99

- building 106
- built-in types 100-106
- change detection mechanism 112, 113
- parameters, passing to 110-112
- syntax 100
- testing 389

used, for manipulating data 99

used, for sorting data 106-109

POST operation 189

predicate function 383

Progressive Web Applications (PWA) 4

promises 158

- rejected parameter 158
- resolve parameter 158

property binding 62

provideHttpClient method 190

provide object literal syntax 147

providers

- overriding, in injector hierarchy 147

pushState 226

PUT operation 189

Q

query parameters

- used, for filtering data 248-250

R

reactive forms 266

- building 271
- form builder, using 285-287
- interacting with 271-275
- modifying, dynamically 278-285
- nesting form hierarchies, creating 276-278
- validation 294-297

reactive programming 155

- in Angular 162-165

ReactiveX library 165

rest parameters 31

rich composition 165

route access

- controlling 252-254

- route configuration** 14
- route data**
- prefetching 256-259
- routed component**
- testing 395, 396
- route parameters** 240
- components, reusing
 - with child routes 245-247
 - input properties, binding to 250, 251
 - query parameters, used for
 - filtering data 248-250
 - snapshot, taking of 247
 - used, for building detail page 240-245
- router**
- testing 395
- router guards**
- testing 398-400
- router links**
- styling 239, 240
- router resolvers**
- testing 401-403
- routing component**
- testing 397
- runtime errors, in Angular application**
- global error handler, creating 312-315
 - handling 308
 - HTTP request errors, catching 308-312
 - responding, to 401 Unauthorized error 315, 316
- rxjs-interop package** 182
- RxJS library** 155, 165
- cooperating with 182-184
 - observables, creating 166
 - observables, transforming 167-169
- S**
- schematics** 6
- Search Engine Optimization (SEO)** 422
- Separation of Concerns (SoC)** 121
- Server-Side Rendering (SSR)** 3, 10, 422
- overriding, in Angular applications 425-427
- service-in-a-service** 140
- services**
- testing 385, 386
 - testing, with dependencies 387-389
- service scope limiting** 138
- setup functionality** 364
- Shadow DOM** 82
- signal-based components** 178
- signals** 177, 178
- Single-Page Applications (SPAs)** 225
- Single Responsibility Pattern (SRP)** 36
- slice pipe** 100
- snackbar component** 357
- applying 357, 358
- sort method** 108
- spread parameter** 30
- spying** 370
- SSG applications**
- prerendering 440, 441
- SSR applications**
- rendering 422-425
- Static Site Generation (SSG)** 10
- string type** 42
- stub**
- dependency, replacing with 371-375
- stubbing** 370
- subscribers** 160

switch statement 69

synchronous methods

testing 387

T

target event 74

target property 62

teardown functionality 364

template-driven forms 266

building 267-270

validation 290-294

template expression 62

template input variable 66

template reference variables 81, 82

template statement 74

ternary operator 34

test spec 363

test suite 363

Time to First Byte (TTFB) 418

transform method

args 107

value 106

tree shaking 130

triggers 439

two-way binding 267

type casting 50

TypeScript 37-39

interfaces 48-50

types 41

working with 39-41

TypeScript language, interfaces

generics 50, 51

utility types 52

TypeScript language, types

any type 43

array type 42

boolean type 42

classes type 45-48

custom types 43, 44

function type 44, 45

number type 42

string type 42

U

UI components, Angular Material

card 341-343

chips 337, 338

data table 344-350

form controls 330

input 330-335

layout 340

navigation 338-340

overlays 350

pop-up dialog 350

select 335-337

unit tests 362, 365

anatomy 363-365

need for 362, 363

test spec 363

test suite 363

utility types 52

V

variable declaration 28-30

viewport 438

reference link 438

views 56

Visual Studio Code (VSCode) 2

VSCode Debugger 20, 21

VSCode profiles 22

Angular Language Service 22, 23
EditorConfig 24, 25
Material Icon Theme 24

W

watch mode 370
web forms 265-267
web-vitals 419
wildcard route 228, 238
writable signals 178-180
 working 179

Z

Zone.js library 85, 313
zone-less applications 178

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835087480>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

