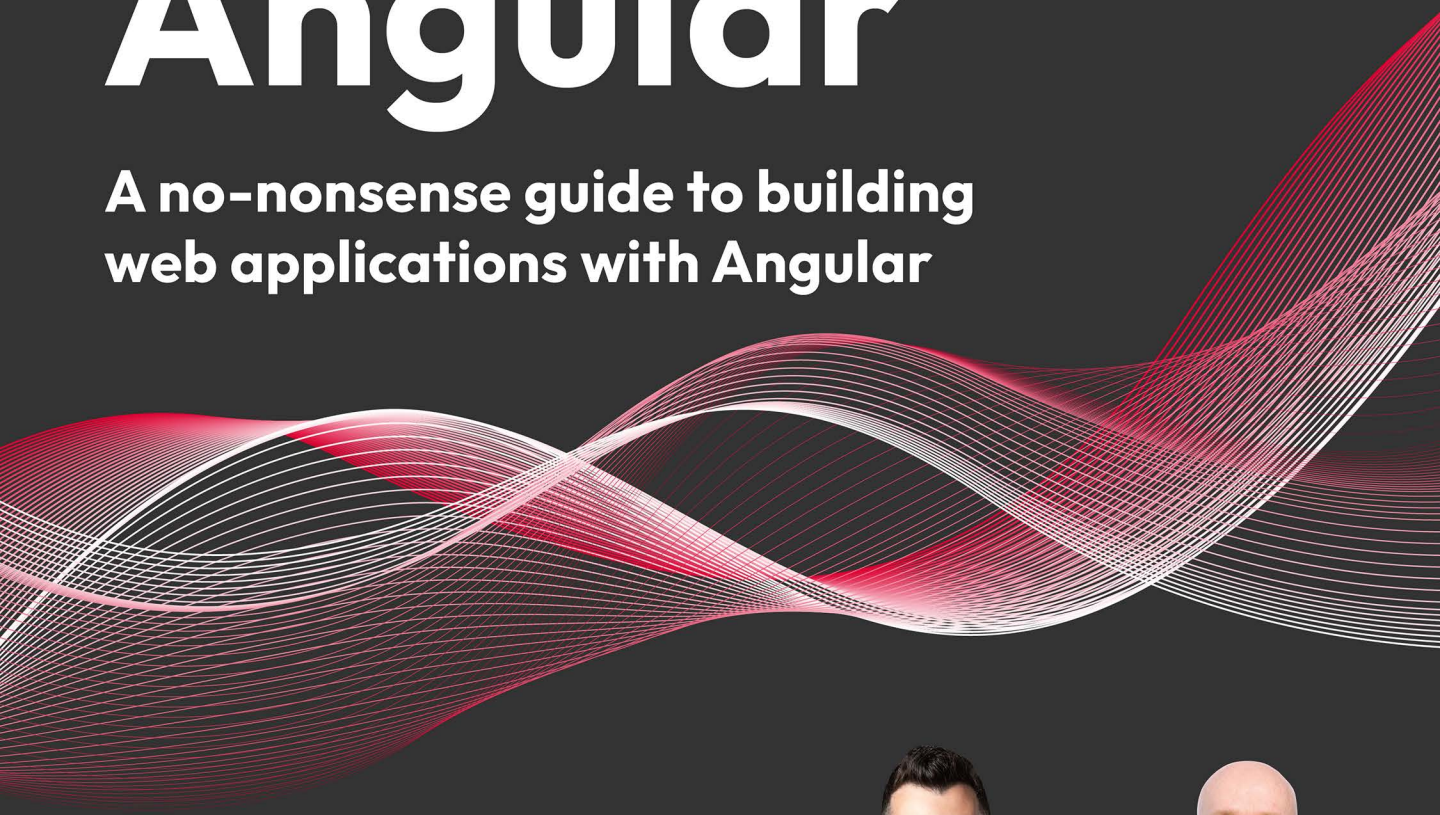
EXPERT INSIGHT

Learning Angular

A no-nonsense guide to building web applications with Angular



Fourth Edition

Aristeidis Bampakos
Pablo Deeleman

packt

Learning Angular

Fourth Edition

A no-nonsense guide to building web applications with Angular

Aristeidis Bampakos

Pablo Deeleman

BIRMINGHAM—MUMBAI

Learning Angular
Fourth Edition

Senior Publishing Product Manager: Suman Sen
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Meenakshi Vijay
Content Development Editor: Shazeen Iqbal
Copy Editor: Safis Editing
Technical Editor: Srishty Bhardwaj
Proofreader: Safis Editing
Indexer: Hemangini Bari
Presentation Designer: Rajesh Shirsath
Developer Relations Marketing Executive: Priyadarshini Sharma

First published: April 2016
Second edition: December 2017
Third edition: September 2020
Fourth edition: February 2023

Production reference: 2230223

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

ISBN 978-1-80324-060-2

www.packt.com

Contributors

About the authors

Aristeidis Bampakos has over 20 years of experience in the software development industry. He is a Greek national who currently works in Athens as a Web Development Team Leader at Plex-Earth, specializing in the development of web applications using Angular.

He studied Computer Technology at the University of Portsmouth and in 2002 he was awarded the degree of Bachelor of Engineering with Second Class Honours (Upper Division). In 2004, he completed his MSc in Telecommunications Technology at Aston University. 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 Angular later on, 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 2 successful book titles about Angular (Learning Angular – 3rd edition and Angular Projects – 2nd edition), as well as an Angular Senior Tech Instructor at Code.Hub, where he nurtures aspiring Angular developers and professionals.

In his spare time, he enjoys being an occasional speaker in meetups, conferences, and podcasts, where he talks about Angular. He is currently leading the effort of making Angular accessible to the Greek development community by maintaining the open-source Greek translation of the official Angular documentation.

To my pet bunny Elpida (Hope), my writing companion in this authoring journey and my best friend during the last 10 years. R.I.P.

Pablo Deeleman has contributed to the dev community with several books on Angular since 2016, all published by Packt Publishing.

With sound expertise in front-end libraries and frameworks such as Backbone.js, Knockout.js, VueJS, React, Svelte, AngularJs, and Angular, Pablo Deeleman has developed his career since 1998 as a JavaScript engineer across a broad range of successful companies, such as Gameloft, Red Hat or Dynatrace, just to name a few. He currently works as Staff Software Engineer at Twilio, the global leader in customer engagement communications.

I'd like to thank my team at Twilio and the awesome bunch of passionate individuals that support and challenge me to become a better professional each day: Gemma Gelida, David Luna, Natalia Venditto, Adrián González, Rafael Marfil, Andreia Leite, and many more. I'd like to personally thank Aristeidis Bampakos for driving this editorial franchise to an unparalleled level of excellence.

About the reviewers

Jurgen Van de Moere is a front-end architect based in Belgium.

He began his career in 1999 and worked for more than a decade as a web developer and system engineer for large companies across Europe. In 2012, driven by his passion for web technologies, Jurgen decided to specialise in JavaScript and Angular.

Since then, he has helped many leading businesses succeed in building secure, maintainable, testable, and scalable Angular applications. In his mission to continually share his knowledge with others, Jurgen serves as a private advisor and mentor to world-renowned businesses and developers around the world.

You won't find Jurgen in the spotlight very often as he loves to spend time with his family, reading books, and writing articles. His writings impact thousands of developers a day and are regularly featured by some of the leading publishers in the tech industry.

Jurgen is actively involved in growing the Belgian Angular community as co-organizer of NG-BE, Belgium's first ever Angular conference. In 2016, he was awarded through the Google GDE program as the first ever Google Developer Expert in Belgium for web technologies.

You can reach Jurgen at `hire@jvandemo.com`, follow him on Twitter at `https://twitter.com/jvandemo` or read his articles on `https://jvandemo.com`.

Santosh Yadav works as a Senior Software Engineer at Celonis and is a GDE for Angular, GitHub Star, and an Auth0 Ambassador. He loves contributing to Angular and its ecosystem. He is co-founder of This is Learning. He is also the author of the Ngx-Builders package and part of the NestJsAddOns core Team. He is also running This is Tech Talks talk show, where he invites industry experts to discuss different technologies. You can find him on Twitter (*@SantoshYadavDev*).

I would like to dedicate this book to my wife, Rekha, and daughter, Hiya. Also, I would like to thank Lars Gyrup, Brink Nielsen, Tanay Pant, Anuj Tripathi, Sajith Karad, and my teammates at Celonis for all the help they provided through the year and for keeping me motivated.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`

Table of Contents

Chapter 12: Unit Test an Angular Application 337

Chapter 13: Bringing an Application to Production 371

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 fourth edition, will show you how you can use it to achieve cross-platform high performance with the latest web techniques, extensive integration with modern web standards, and **integrated development environments (IDEs)**.

The 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, deploy them to a hosting provider.

Updated for Angular 15, this new edition covers lots of new features and practices that address the current frontend web development challenges. You'll find a new dedicated chapter on observables and RxJS, 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 JavaScript and full-stack developers dipping their feet for the first time in to the world of frontend development with Angular, as well as those migrating to the Angular framework to build professional web applications. You'll need prior exposure to JavaScript and a solid foundation in the basics of web programming before you get started with this book.

What this book covers

Chapter 1, *Building Your First Angular Application*, shows how to set up the development environment by installing the Angular CLI and explains how to use schematics (commands) to automate tasks such as code generation and application building.

We will create a new simple application using the Angular CLI and build it. We will also learn about some of the most useful Angular tools that are available on Visual Studio Code.

Chapter 2, *Introduction to TypeScript*, explains what is TypeScript, the language that is used when creating Angular applications, and covers the most basic building blocks, such as types, template strings, lambdas, and classes. We will learn how to use decorators that are widely used in Angular classes and modules. We will take a look at some of the advanced types and the latest features of the language.

Chapter 3, *Organizing Application into Modules*, explains what modules are in Angular and how they differ from modules in TypeScript. We will learn about the most common modules that we use in Angular and discuss the purposes of the different types of modules.

Chapter 4, *Enabling User Experience with Components*, explains how a component is connected to its template and how to use a TypeScript decorator to configure it. We will 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 for detecting changes in a component. We will also learn how to create standalone components.

Chapter 5, *Enrich Applications Using Pipes and Directives*, covers the built-in directives and pipes. We will build our own custom pipe and directive and use them in a sample application that demonstrates their use. We will also learn the difference between attribute and structural directives.

Chapter 6, *Managing Complex Tasks with Services*, explains how the dependency injection mechanism works, how to create and use services in components, and how we can create providers in an Angular application.

Chapter 7, *Being Reactive Using Observables and RxJS*, discusses reactive programming and how we can use observables in the context of an Angular application through the RxJS library. We will also take a tour of all the common RxJS operators that are used in an Angular application.

Chapter 8, *Communicating with Data Services over HTTP*, explains how to interact with a remote backend API and perform CRUD operations with data in Angular. We will also investigate how to set additional headers to an HTTP request and intercept such a request to act before sending the request or upon completion.

Chapter 9, *Navigate through Application with Routing*, explains how to use the Angular router in order to activate different parts of an Angular application. We will find out how to pass parameters through the URL and how we can break an application into routing modules that can be laz ily loaded. We will 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*, explains how to use Angular reactive forms in order to integrate HTML forms and how to set them up using FormGroup and FormControl. We will track the interaction of the user in the form and validate input fields.

Chapter 11, *Introduction to Angular Material*, discusses how to integrate Google Material Design guidelines in to an Angular application using a library called Angular Material that is developed by the Angular team. We will take a look at some of the core components of the library and their usage. We will discuss the themes that are bundled with the library and how to install them.

Chapter 12, *Unit Test an Angular Application*, explains how to test Angular artifacts and override them in a test, examines the different parts of a test, and explains which parts of a component should be tested.

Chapter 13, *Bringing Application to Production*, discusses the hosting providers that are supported by the Angular CLI. We will perform build optimizations prior to deployment, and we will use the Angular CLI to deploy to GitHub Pages.

Chapter 14, *Handling Errors and Application Debugging*, explains how to handle different types of errors in an Angular application and understand errors that come from the framework itself. We will learn how to debug an Angular application using and how to profile it using Angular DevTools.

To get the most out of this book

- You will need a version of Angular 15 installed on your computer, preferably the latest one. All code examples have been tested using Angular 15.0.0 on Windows, but they should work with any future release of Angular 15 as well.
- If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available 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

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Learning-Angular-Fourth-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/af51s`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "When we run the preceding command in the `src\app\products` folder, it will create the `sort.pipe.ts` file and its corresponding unit test file, `sort.pipe.spec.ts`."

A block of code is set as follows:

```
let distance: any;
distance = '1000km';
distance = 1000;
const distances: any[] = ['1000km', 1000];
```

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

```
const basket: any = new FruitBasket();
```

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

```
ng generate interface product
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "There are two categories of pipes: **pure** and **impure**."

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 reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click Submit Errata, and fill in the form.

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`.

Share your thoughts

Once you've read *Learning Angular, Fourth 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/9781803240602`

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

To better understand how to develop an Angular application, we need to learn some basic but essential things to have a great experience on our journey with the Angular framework. One of the primary things we should know is what Angular is and why we should start using it for web development. We will also take a tour of the history of Angular to understand how the platform evolved.

Another important but sometimes painful topic is setting up our development environment. It must be done at the beginning of a project but getting this right early can reduce friction as our application grows. Therefore, a large part of this chapter is dedicated to **Angular CLI**, a tool developed by the Angular team that provides scaffolding and automation tasks in an Angular app, eliminating configuration boilerplate and saving developers from facing future frustrations. 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 in an Angular project without 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 for debugging and building our app. We will highlight **Visual Studio Code (VS Code)**, one of the most popular editors in the Angular ecosystem, with a rich collection of extensions for working with Angular.

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
- Structure of Angular application
- VS Code tooling

Technical requirements

- **GitHub**: `https://github.com/PacktPublishing/Learning-Angular-Fourth-Edition/tree/main/ch01`
- **Node.js**: `https://nodejs.org`
- **Git**: `https://git-scm.com/downloads`
- **VS Code**: `https://code.visualstudio.com/download`

What is Angular?

Angular is a development platform that is written in the **TypeScript** language. It consists of smaller sub-systems, including a **JavaScript** framework, a command-line interface, a language service, and a rich collection of first-party libraries.

Angular enables developers to build scalable web applications with TypeScript, a strict syntactic superset of JavaScript. Developing with Angular does not require knowledge of JavaScript, but it is nice to have. We will learn more details about TypeScript in *Chapter 2*, *Introduction to TypeScript*.

The official Angular documentation can be found at `https://www.angular.io`.

We suggest relying first on the official Angular documentation and then on any other sources because it is the most up-to-date source for Angular development.

Angular was created by a team internally at Google. 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. The team joined forces with the TypeScript team at Microsoft and introduced TypeScript into the framework. A vital consideration towards that decision was **decorators**, a powerful feature of the TypeScript language that Angular heavily uses. The next version of the framework, 2.0, was written in TypeScript and re-branded as *Angular* with a different logo than AngularJS.

In this book, we will cover **Angular 15**, which is the latest *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. It is compatible with the two most recent major versions of all browsers except for Chrome and Firefox, which supports the latest ones.

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

Why choose Angular?

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

- The main pillars of the platform: cross-platform, incredible tooling, and easy onboarding
- The usage of Angular worldwide

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 JavaScript framework. However, it is open-source and is backed by a vast and incredible community that enables the framework to run on the remaining three using the following integrations:

- **Angular Universal**: Renders Angular applications server-side
- **Angular Service Worker**: Enables Angular applications to run as **Progressive Web Applications (PWA)** that are customizable and can be installed on a desktop environment
- **Ionic Framework**: 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 creation to deployment.
- **Angular DevTools**: A browser extension that enables us to debug and profile Angular applications from the comfort of our browser. We will learn more about this tool in *Chapter 14*, *Handling Errors and Application Debugging*.

The Angular CLI is the de facto solution for working 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 for a web developer 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:

- Angular HTTP client to communicate with a REST API endpoint over HTTP
- Angular forms to create HTML forms for collecting input and data from users
- Angular router to perform in-app navigations

A first-party library is a library that is provided from the Angular framework out of the box without the need to install it separately.

The preceding libraries are installed by default when we create a new Angular application using the Angular CLI. However, they are not used in our application unless we import them explicitly into our project.

Who uses Angular?

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.

Statistically, more than 2,500 projects inside Google use the Angular framework. Additionally, more than 1.5 million developers worldwide prefer Angular for web development.

The fact that Angular is already used internally at Google is a crucial factor for the reliability of the platform. Every new version of Angular is first 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 platform to the rest of the developer community.

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

Setting up the Angular CLI workspace

Setting up a frontend project today is more cumbersome than ever. We used to manually include the necessary JavaScript and **CSS** files in our **HTML**. Life used to be simple. Then frontend development became more ambitious: we started splitting our code into modules and using special tools called **preprocessors** for our code and CSS.

Our projects became more complicated, and we started to rely on build systems to bundle our applications. As developers, we are not huge fans of configuration—we want to focus on building awesome apps. However, modern browsers do more to support the latest web standards, and some have even started to support JavaScript modules. That said, this is far from being widely supported. In the meantime, we still have to rely on bundling and module support tools.

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. Scaffolder tools almost become necessary as complexity grows and where every hour counts towards 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, and create a production-grade bundle. The Angular CLI supports all of this with the use of special commands.

Prerequisites

Before we begin, we need to ensure that our development environment includes a set of 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` in the command line to check which version you are running. The Angular CLI uses Node.js to accomplish specific tasks, such as serving, building, and bundling your application.

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 that are 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 for creating Angular projects, we need to install it globally in our system. Open a terminal and run the following command:

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

On some Windows systems, you may need elevated permissions, so you should run your terminal as an administrator. In Linux/macOS systems, run the command using the `sudo` keyword.

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

- `install` or `i`: Denotes the installation of a package
- `-g`: 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 major version as the Angular framework, which in this book is 15. 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 **Angular CLI 15**, you can run the following command:

```
npm install -g @angular/cli@15
```

The preceding command will fetch and install the latest version of Angular CLI 15.

CLI commands

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

```
ng command [options]
```

Here, the `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 also be invoked using an alias instead of the actual command name. In this book, we revise 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 (s)`: Builds an Angular application and serves it using a pre-configured web server.
- `test (t)`: Runs unit tests of an Angular application.
- `deploy`: Deploys an Angular application to a web-hosting provider. You can choose from a collection of providers included in the Angular CLI.
- `add`: Installs an Angular library to an Angular application.
- `completion`: Enables auto-complete for Angular CLI commands through the terminal.
- `update`: Updates an Angular application to the latest Angular version.

Updating an Angular application is one of the most critical tasks from the preceding list. It helps us stay updated by upgrading our Angular applications to the latest platform 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 that contains tips and step-by-step instructions on how to update your application at `https://update.angular.io`.

Creating a new project

Now that we have prepared our development environment, we can start creating magic by scaffolding our first Angular application. We use the `new` command of the Angular CLI and pass the name of the application that we want to create as an option. To do so, go to a folder of your choice and type the following:

```
ng new my-app
```

Creating a new Angular application is a straightforward process. The Angular CLI will ask you for details about the application you want to create so that it can scaffold the Angular project as best as possible. Initially, it will ask if you 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.io/analytics. (y/N)
```

The Angular CLI will only ask the previous question once when you create your first Angular project and apply it globally in your system. However, you can change the setting later in a specific Angular workspace. Next, it will ask if you want to include routing in your application:

```
Would you like to add Angular routing? (y/N)
```

Routing is related to navigating from one view of your application to another, which we will learn later in *Chapter 9*, *Navigate through Application with Routing*. For now, answer `No` to the question and press *Enter*. The next question is related to the styling of your application:

```
Which stylesheet format would you like to use? (Use arrow keys)
```

It is common to use CSS for styling Angular applications. However, you can use preprocessors like **SCSS** or **Less** to add value to your development workflow. In this book, we work with CSS directly, so accept the default choice, `CSS`, and press *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, test, and publish the Angular application:

- `.vscode`: Includes VS Code configuration files
- `node_modules`: Includes npm packages needed for development and running the Angular application
- `src`: Contains the source files of the application
- `.editorconfig`: Defines coding styles for your editor
- `.gitignore`: Specifies 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`: TypeScript configuration that is specific to the Angular application
- `tsconfig.json`: TypeScript configuration that is specific to the Angular CLI workspace
- `tsconfig.spec.json`: TypeScript configuration that is specific to unit tests

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

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

```
ng serve
```

Remember that any Angular CLI commands must be run inside an Angular CLI workspace folder.

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`:

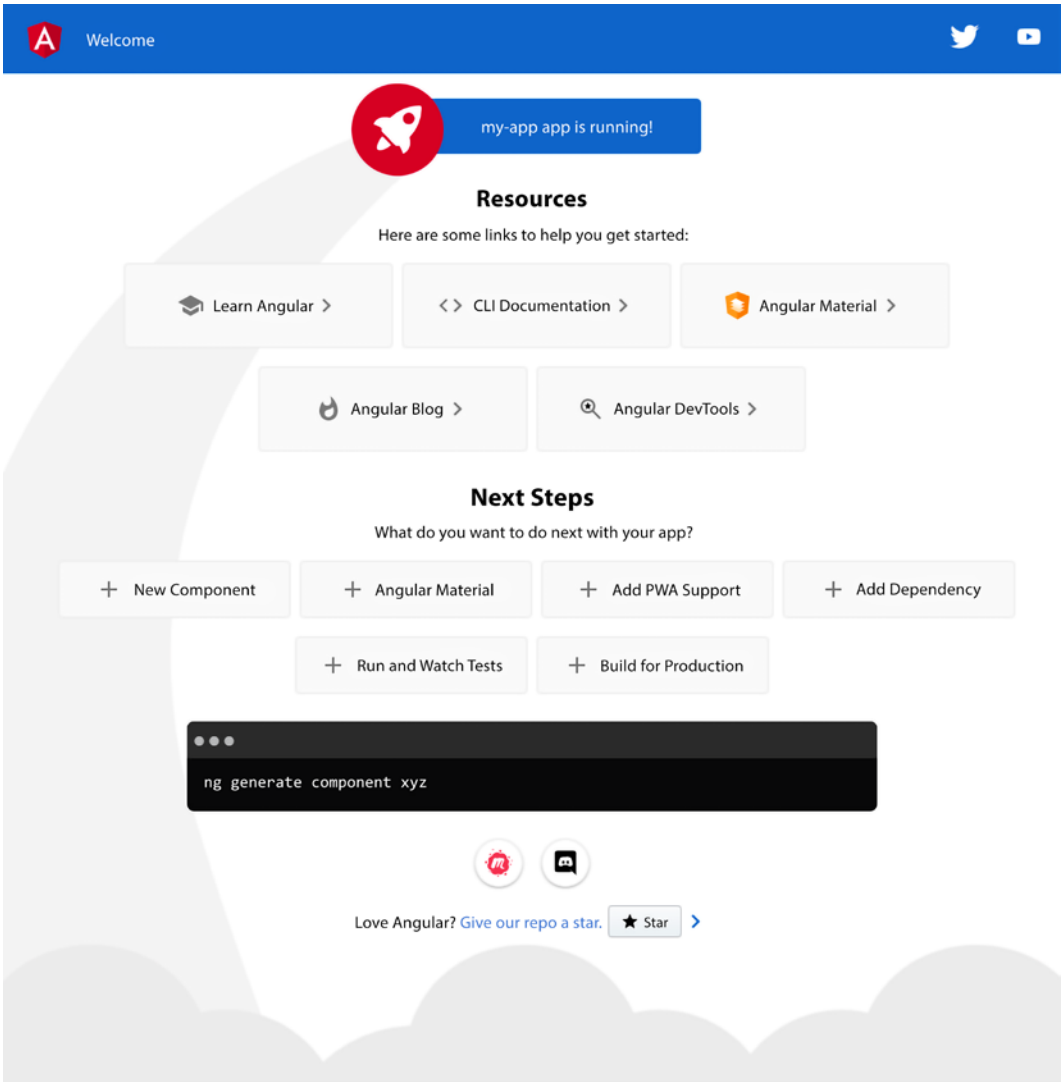


Figure 1.1: Angular application landing page

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

Structure of an Angular application

We are about to take the first intrepid steps into examining our Angular application. The Angular CLI has already scaffolded our project and has carried much 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 VS Code in this book, but feel free to choose any editor you are comfortable with:

1. Open VS Code and select **File | Open Folder...** from the main menu.
2. Navigate to the `my-app` folder and select it. VS Code will load the associated Angular project.
3. Navigate to 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 is also where we define the styles of our application and any static assets we use, such as icons, images, and JSON files. It contains the following:

- `app`: Contains all the Angular-related files of the application. You interact with this folder most of the time during development.
- `assets`: Contains static assets such as fonts, images, and icons.
- `favicon.ico`: The icon displayed in the tab of your browser, along with the page title.
- `index.html`: The main HTML page of the Angular application.
- `main.ts`: The main entry point of the Angular application.
- `styles.css`: Contains application-wide styles. These are 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 is created automatically from the Angular CLI contains the following files:

- `app.component.css`: Contains CSS styles specific for the sample page
- `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.module.ts`: Defines the main **module** of the Angular application

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

In the following sections, we will learn about the purpose of each of those files in more detail.

Components

The files whose name starts 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**. A typical Angular application has at least a main component called `AppComponent` by convention.

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 4*, *Enabling User Experience with Components*.

When the Angular CLI builds an Angular app, it first parses the `index.html` file and starts identifying HTML tag elements inside the `body` tag. An Angular application is always rendered inside the `body` tag 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 which components belong to the app?

Modules

The main module of our application is a TypeScript file that acts as a container for the main component. The component was registered with this module upon creating the Angular application; otherwise, the Angular framework would not be able to recognize and load it. A typical Angular application has at least a main module called `AppModule` by convention.

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

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

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

The main task of the bootstrapping file is to define the module that will be loaded at application startup. It calls the `bootstrapModule` method of the `platformBrowserDynamic` method and passes `AppModule` as the entry point of the application.

As we have already learned, Angular can run on different platforms. The `platformBrowserDynamic` method that we use targets the browser platform.

We will learn more about the capabilities of Angular modules in *Chapter 3*, *Organizing Application into Modules*.

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 to start the application:

   ```
   ng serve
   ```

2. Open the application with your browser at `http://localhost:4200` and notice the text next to the rocket icon that reads **my-app app is running!** The word **my-app** that corresponds to the name of our application 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';

   @Component({
   ```

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

The `title` variable is a **property** of the component and is currently used in the component template.

3. Open the `app.component.html` file and go to line 344:

```
<span>{{ title }} app is running!</span>
```

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 displays it on the page.

Angular uses template syntax to extend and enrich the standard HTML syntax in the template of an application. We will learn more about the syntax used in Angular templates in *Chapter 4*, *Enabling User Experience with Components*.

4. Change the value of the `title` property in the TypeScript class to `Learning Angular` and examine the application in the browser:

Figure 1.2: Landing page title

Congratulations! You have successfully used the Angular CLI to interact with the Angular application.

By now, you should have a basic understanding of how Angular works and what are the basic parts of a sample Angular application. As a reader, you had to swallow much information at this point. However, you will get a chance to get more acquainted with the components and modules 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 just a few steps are needed to display an application on the screen.

VS Code tooling

VS Code is an open-source code editor backed by Microsoft. It is very popular in the Angular community, primarily because of its robust support for TypeScript. TypeScript has been, to a great extent, a project driven by Microsoft, so it makes sense that one of its popular editors was conceived with built-in support for this language. All the nice features we might want are already baked in, including syntax, error highlighting, and automatic builds. Another reason for its broad popularity is the various extensions available in the marketplace that enrich the Angular development workflow. What makes VS Code so great is not only its design and ease of use but also the access to a ton of plugins, and there are some great ones for Angular development. The most popular are included in the **Angular Essentials** extension pack from John Papa. To get it, go through the following steps:

1. Navigate to the **Extensions** menu of VS Code.
2. Type `Angular Essentials` in the search box.
3. Click the **Install** button on the first entry item.

Alternatively, you can install it automatically since it is already included in the repository of this book's source code. When you download the project from GitHub and open it in VS Code, it will prompt you to view and install the recommended extensions:

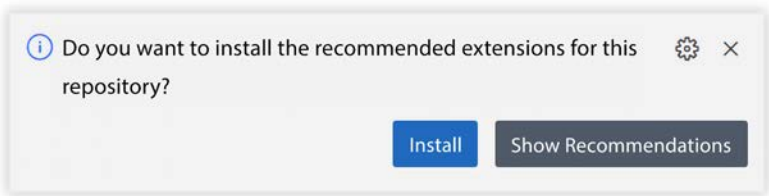


Figure 1.3: Recommended extensions prompt

In the following sections, we will look at some of the extensions included in the Angular Essentials pack as well as other useful extensions for Angular development.

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 VS Code with the following features:

- Code completion
- 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 = 'Learning Angular';
  description = 'Hello World';
}
```

2. Open the `app.component.html` file and start typing the name of the property in the template. The Angular Language Service will find it and suggest it for you automatically:

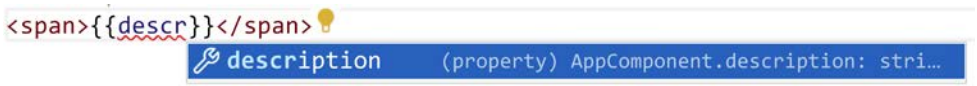


Figure 1.4: Angular Language Service

The `description` property is a *public* property. In public methods and properties, we can omit the keyword `public`. Code completion works only for public 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 were typing. 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:

Figure 1.5: 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.

Angular Snippets

The **Angular Snippets** extension contains a collection of TypeScript and HTML code snippets for various Angular artifacts, such as components. To create the TypeScript class of an Angular component using the extension, go through the following steps:

1. Open our Angular application in VS Code and click on the **New File** button next to the workspace name in the **EXPLORER** pane.
2. Enter a proper file name, including the `.ts` extension, and press *Enter*.
3. Type `a-component` inside the file and press *Enter*.

The extension builds the TypeScript code for you automatically:

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

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

export class NameComponent implements OnInit {
  constructor() { }

  ngOnInit() { }
}
```

There are also other available snippets for creating various Angular artifacts, such as modules. All snippets start with the `a-` prefix.

Nx Console

Nx Console is an interactive UI for the Angular CLI that aims to assist developers who are not very comfortable with the command-line interface or do not want to use it. It works as a wrapper over Angular CLI commands, and it helps developers concentrate on delivering outstanding Angular applications instead of trying to remember the syntax of every CLI command they want to use.

The extension is automatically enabled when you open an Angular CLI project. If you click on the Nx Console menu of VS Code, it displays a list of Angular CLI commands that you can execute:

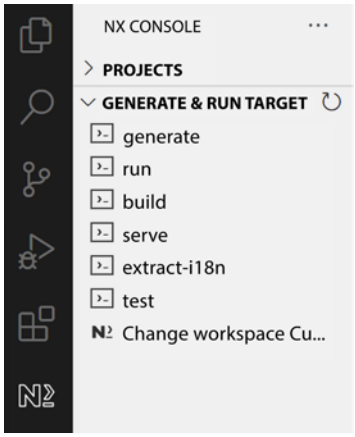


Figure 1.6: Nx Console

Nx Console includes most of the Angular CLI commands. The user interface allows developers to use them without remembering the available options each command supports.

Material icon theme

VS Code has a built-in set of icons that it uses to display different types of files in a project. This extension provides additional icons that conform to the Material Design guidelines by Google; a subset of this collection targets Angular-based artifacts.

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

EditorConfig

VS Code editor settings, such as indentation or spacing, can be set at a user or project level. **EditorConfig** can override these settings using a configuration file called `.editorconfig`, which can be found in the root folder of an Angular CLI project. You can define unique settings in this file to ensure the consistency of the coding style across your team.

Angular Evergreen

The stability of the Angular platform is heavily based on the regular release cycles according to the following schedule:

- A major version every six months

- Various minor versions between each major one
- A patch version every week

The **Angular Evergreen** extension helps us follow the previous schedule and keep our Angular projects up to date. It provides us with a unique user interface containing information about the current Angular version of our project and actions we can take to update it. Updating an Angular project can also be accomplished using a custom menu that appears if we right-click on the `angular.json` file of the workspace.

In a nutshell, Angular Evergreen is loaded in an Angular CLI project and compares the local version of Angular with the latest one. If there is a new version, it will alert the user by displaying a notification message:

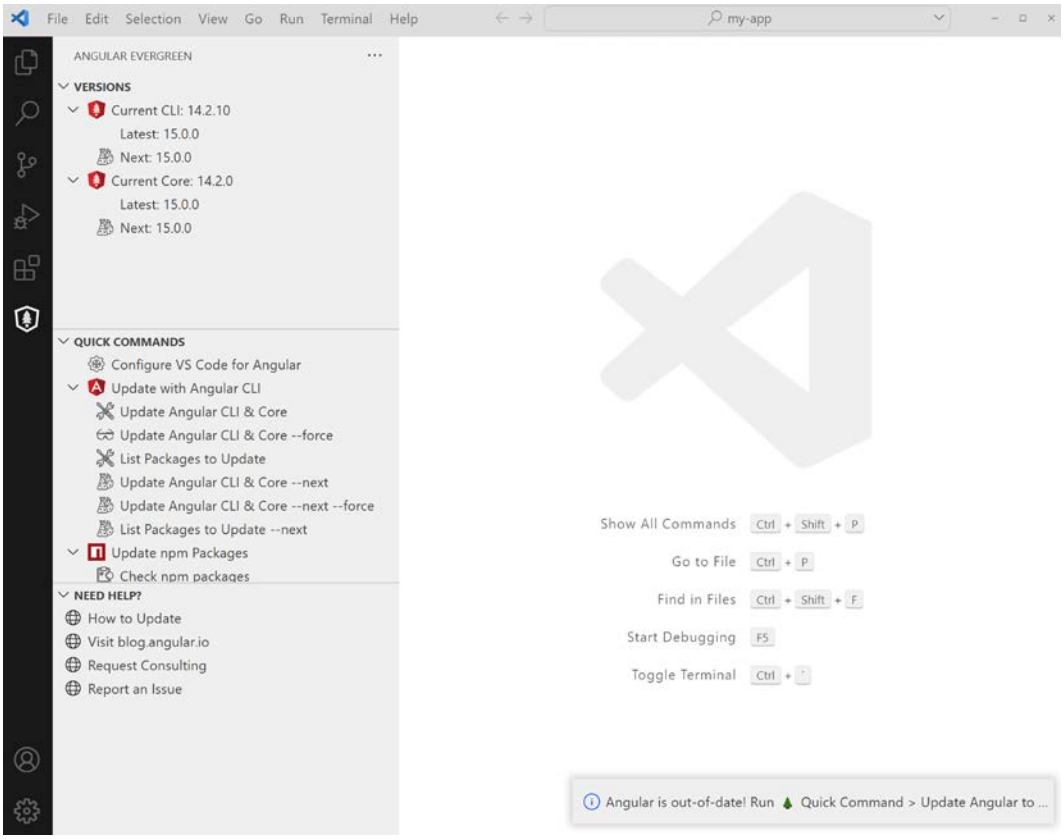


Figure 1.7: Angular Evergreen

Additional to the latest version, we can update to the next Angular version, which is the beta version currently under development, or upgrade other npm packages of the project.

Rename Angular Component

Renaming Angular artifacts such as components and modules usually requires visiting different parts of the codebase in a project. Working your way through the project when you are a beginner is a daunting task. Thankfully, there is an extension that can help us with that.

The **Rename Angular Component** extension provides a user-friendly interface that enables us to rename certain Angular artifacts. We can right-click on any component-related file, select the **Rename Angular Component** option and enter the new name of the component. The extension will find all template and TypeScript files that use the component and rename them accordingly.

Although the name of the extension refers to components, it can currently be used with other Angular artifacts such as services, directives, and guards. We will learn about these artifacts later in the book.

Summary

That's it! Your journey to 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 platform, and examined the benefits of using it for web development.

We saw how to set up our development workspace and find the tools we need to bring TypeScript into the game. We introduced the Angular CLI tool, the Swiss Army knife for Angular, that 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 main module of an Angular application and looked at how Angular bootstraps it at application startup.

Finally, we met the VS Code editor and learned how it could empower you as a software developer. We examined some of the essential available plugins and extensions for Angular that save quite a few keystrokes. Your focus and energy should be spent on solving the problem and structuring your solution, not making your fingers tired. We encourage you to learn more about your editor and its possibilities because this will make you faster and more efficient.

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.

2

Introduction to TypeScript

As we learned in the previous chapter, where 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 and study them in detail on our journey through Angular. The good news is that TypeScript is not all that complex, and we will manage to cover most of its relevant parts.

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

- The history of TypeScript
- Types
- Functions, lambdas, and execution flow
- Common TypeScript features
- Decorators
- Advanced types
- Modules

We will first investigate the background of TypeScript and the rationale behind its creation. We will also learn what tools and online resources are available to practice with TypeScript. We will emphasize the typing system, which is the main advantage of TypeScript, and learn how we can use it to create some basic types. We will expand our typing knowledge by learning how to use classes, interfaces, and advanced types in the Angular context. At the end of the chapter, we will explore how to organize the structure of an application by combining the typing system with modules.

The history of TypeScript

Transforming small web applications into thick monolithic clients was impossible due to the limitations of earlier JavaScript versions, such as the **ECMAScript** 5 specification. 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 that never seemed to fit the bill.

As a response to these concerns, ECMAScript 6 (also known as ES6 or ES2015) promised to solve these issues by introducing better module loading functionalities, an improved language architecture for better scope handling, and a wide variety of syntactic sugar to better manage types and objects. Class-based programming introduced an opportunity to embrace a more **Object-Oriented Programming** (**OOP**) approach when building large-scale applications.

Microsoft took on this challenge and spent nearly two years building a superset of the language, combining the conventions of ES6 and borrowing some proposals from the next specification version. The idea was to launch something that would help build enterprise applications with a lower error footprint using static type checking, better tooling, and code analysis. After two years of development led by Anders Hejlsberg, lead architect of C# and creator of Delphi and Turbo Pascal, TypeScript 0.8 was finally introduced in 2012 and reached version 1.0 2 years later. It was not only running ahead of ES6 but also implemented the same features and provided a stable environment for building large-scale applications. It introduced, among other features, 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 benefits of TypeScript

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 advocating TypeScript in your next project, and all of them apply to Angular as well.

Here is a short rundown, to name a few:

- Annotating your code with types ensures a 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 gets minimized in the long run.
- TypeScript extends classes with long-time demanded features such as class fields, private members, and enumerations.
- Decorators allow you to extend your classes and implementations in unique ways.
- Interfaces ensure a smooth and seamless integration of your libraries in other systems and code bases.
- TypeScript support across different IDEs is terrific, 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 coming from other OOP-based backgrounds such as Java, C#, and C++.

Introducing TypeScript resources

Let's have a look at where we can get further support to learn and test-drive our new knowledge of TypeScript.

In this book, we will be using TypeScript 4.8 as it is supported by Angular 15.

The official website

Our first stop is the official website of the language 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. We encourage you to leverage this tool to test the code examples we cover throughout this chapter.

The official wiki documentation

The code for TypeScript is fully open-sourced at GitHub, and the Microsoft team has put reasonable effort into documenting the different facets of the code in the wiki available on the repository site. We encourage you to take a look at it any time you have a question or if you want to dive deeper into any of the language features or form aspects of its syntax. The wiki is located at `https://github.com/Microsoft/TypeScript/wiki`.

In the following section, we will introduce the typing system of TypeScript. We will explore the most basic types of the TypeScript language. We will also learn how to benefit from the typing system and create custom and dynamic types to enhance our applications further.

Types

Working with TypeScript or any other coding language means working with data, and such data can represent different sorts of content that are called **types**. Types are used to represent the fact that such data can be a text string, an integer value, or an array of these value types, among others. You may have already met types in JavaScript since we have always worked implicitly with them. This also means that 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 it to return type-wise. We can enforce this flexibility using a specific type called `any`, as we will see later in this chapter. However, statically typing our variables gives our IDE and us a good picture of what kind of data we are supposed to find in each instance of code. 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 a piece of text:

```
var brand: string = 'Chevrolet';
```

Check out the type definition next to the variable name, separated by a colon. It is the way to annotate types in TypeScript.

We can use single or double quotes for the value of a string variable. Feel free to choose either and stick with it within your team. We can define multiline text strings with support for text interpolation using placeholder variables and backticks:

```
var brand: string = 'Chevrolet';
var message: string = 'Today it's a happy day! I just bought a new
${brand} car';
```

In the preceding snippet, any variables that we may use inside the multiline text must be surrounded by the curly braces of the placeholder `${}`.

Declaring variables

TypeScript, as a superset of JavaScript, supports expressive declaration nouns such as `let`, which denotes that the scope of the variable is the nearest enclosing block (either a function, `for` loop, or any other enclosing statement). On the other hand, `const` indicates that the value of the declared variable cannot be changed once it's set.

The let keyword

Traditionally, developers have been using the keyword `var` to declare objects, variables, and other artifacts, but this is discouraged when you start using ES6 or TypeScript. The reason is that ES5 only has a function scope; that is, a variable is unique within the context of a function:

```
function test() {
    var x;
}
```

There can be no other variable declared as `x` in this function. If you do declare one, then you effectively redefine it. However, there are cases in which scoping is not applied, such as in loops. For example, in Java, you would write the following and ensure that a variable will never leak outside of the loop:

```
var i = 3;
for (var i = 0; i < 10; i++) {
}
```

In the preceding snippet, the `i` variable outside the loop will not affect the `i` variable inside it because they have a different scope. To overcome this limitation, ES6 introduced the `let` keyword:

```
let i = 3;
for (let i = 0; i < 10; i++) {
}
```

So, remember, no more `var`; use the `let` keyword wherever possible.

The const keyword

The `const` keyword is a way to indicate 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 through compile-time support. Consider the following code snippet:

```
const PI = 3.14;
PI = 3;
```

If we try to run it with TypeScript, the compiler will throw the following error message:

```
Cannot assign to 'PI' because it is a constant
```

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 obj = {
    a: 3
};
obj.a = 4;
```

If we declare the `obj` variable as a constant, it does not prevent the entire object from being edited but rather its reference. So, the preceding code is valid. If we try to change the reference of the variable such as `obj = {}`, it is not allowed, and we get the same compiler error.

Prefer to use the `const` keyword when you are sure that the properties of an object will not change during its lifetime. It prevents the object from accidentally changing and enforces data **immutability**, a hot topic in Angular applications.

Number

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

```
const age: number = 7;
const height: number = 5.6;
```

It defines a floating-point number and hexadecimal, decimal, binary, and octal literals.

Boolean

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

```
const isZeroGreaterThanOne: boolean = false;
```

The result of the variable represents the fulfillment of a `boolean` condition.

Array

The **array** type defines a list of items that contain certain types only. Handling exceptions that arise from errors such as assigning wrong member types in a list can now be easily avoided with this type.

The syntax requires the postfix `[]` in the type annotation, as follows:

```
const brands: string[] = ['Chevrolet', 'Ford', 'General Motors'];
const ages: number[] = [8, 5, 12, 3, 1];
```

If we try to add a new item to the `ages` array with a type other than a number, the runtime type-checker will complain, making sure our typed members remain consistent and that our code is error-free.

Dynamic typing with no type

Sometimes, it is hard to infer the data type from the information we have at any given point, especially when we are porting legacy code to TypeScript or integrating loosely typed third-party libraries and modules. TypeScript supplies us with a convenient type for these cases. The `any` type is compatible with all the other existing types, so we can type any data value with it and assign any value to it later:

```
let distance: any;
distance = '1000km';
distance = 1000;
const distances: any[] = ['1000km', 1000];
```

However, this great power comes with great responsibility. If we bypass the convenience of static type checking, we are opening the door to type errors when piping data through our modules. 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 Animal = 'Cheetah' | 'Lion';
```

It is essentially a type with a finite number of allowed values. Let's create a variable of this type:

```
const animal: Animal = 'Cheetah';
```

The preceding code is perfectly valid as `Cheetah` is one of the allowed values and works as intended. The interesting part happens when we give our variable a value it does not expect:

```
const animal: Animal = 'Turtle';
```

The preceding code will result in the following compiler error:

```
Type '"Turtle"' is not assignable to type 'Animal'
```

Enum

The enum type is a set of unique numeric values that we can represent by assigning user-friendly names to each one. Its use goes beyond assigning an alias to a number. We can use it to list the variations that a specific type can assume in a convenient and recognizable way. It begins numbering members, starting at 0 unless explicit numeric values are assigned to them:

```
enum Brands { Chevrolet, Cadillac, Ford, Buick, Chrysler, Dodge };
const myCar: Brands = Brands.Cadillac;
```

In the preceding code, if we inspect the variable `myCar`, we will see that it returns the value 1, which is the index of `Cadillac`. As we mentioned already, we can also assign custom numeric values like the following:

```
enum BrandsReduced { Tesla = 1, GMC, Jeep };
const myTruck: BrandsReduced = BrandsReduced.GMC;
```

In the preceding code, if we inspect the variable `myTruck`, we will see that it returns the value 2 because the first enumerated value, `Tesla`, was set to 1 already. We can extend value assignation to all members as long as such values are integers:

```
enum StackingIndex {
    None = 0,
    Dropdown = 1000,
    Overlay = 2000,
    Modal = 3000
};
const mySelectBoxStacking: StackingIndex = StackingIndex.Dropdown;
```

One last point worth mentioning is the possibility to look up a member mapped to a given numeric value:

```
enum Brands { Chevrolet, Cadillac, Ford, Buick, Chrysler, Dodge };
const myCarBrandName: string = Brands[1];
```

In the preceding snippet, the `myCarBrandName` variable will be equal to `Cadillac`.

It should also be mentioned that from TypeScript 2.4 and onward, it is possible to assign string values to enums. It is a technique preferred in Angular projects because of its extended support in template files.

Void

The `void` type represents the absence of a type, and its use is constrained to annotating functions that do not return an actual value:

```
function test(): void {
    const a = 0;
}
```

In the preceding snippet, there is no return type in the function.

Type inference

Typing is optional since TypeScript is smart enough to infer the data types of variables and function return values out of context with a certain level of accuracy. If it is not possible, it will assign the dynamic `any` type to the loosely typed data at the cost of reducing type checking to a bare minimum.

In the following section, we will embark on a new journey through TypeScript to learn more about TypeScript functions and their execution flow.

Functions, lambdas, and execution flow

Functions are the processing machines we use to analyze input, digest information, and apply the necessary transformations to data. Data can be provided either to transform the state of our application or to return an output that will be used to shape our application's business logic or user interactivity.

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.

Annotating types in functions

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

```
function sayHello(name: string): string {
    return 'Hello, ' + name;
}
```

There are two main differences from the usual function syntax in regular JavaScript. First, we annotate 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. In addition to this, we also annotate the returning value by adding the `string` type to the function declaration.

As mentioned in the previous section, the TypeScript compiler is smart enough to infer types when no annotation is provided. In this case, the compiler looks into the arguments provided and returns statements to infer a returning type from them.

Functions in TypeScript can also be represented as expressions of anonymous functions, where we bind the function declaration to a variable:

```
const sayHello = function(name: string): string {
    return 'Hello, ' + name;
}
```

However, there is a downside to that approach. Although typing function expressions this way is allowed, thanks to type inference, the compiler is missing the type definition of the declared variable. We might assume that the inferred type of a variable that points to a function typed as a string is a string. Well, it's not. A variable that points to an anonymous function ought to be annotated with a `function` type:

```
const sayHello: (name: string) => string = function(name: string): string
{
    return 'Hello, ' + name;
}
```

The `function` type informs us of the types expected in the payload and the type returned by the function execution, if any. This whole block, which is of the form *(arguments: type) => returned type*, becomes the type annotation that our compiler expects.

Function parameters in TypeScript

Due to the type checking performed by the compiler, function parameters require special attention in TypeScript.

Optional parameters

Parameters are a core part of the type checking applied by the TypeScript compiler. Parameters are defined as optional by adding the character `?` after the parameter name:

```
function greetMe(name: string, greeting?: string): string {
```

```
    if (!greeting) {
        greeting = 'Hello';
    }
    return greeting + ', ' + name;
}
```

To call the previous function, we can omit the second parameter in the function call:

```
greetMe('John');
```

So, an optional parameter is not set unless you explicitly pass it a value. It is more of a construct so that you can get help deciding what parameters are mandatory and which ones are optional. To exemplify this, consider the following function signature:

```
function add(mandatory: string, optional?: number) {}
```

We can invoke the previous function in the following ways:

```
add('some string');
add('some string', 3.14);
```

Both versions are valid. Be aware that optional parameters should be placed last in a function signature. Consider the following function:

```
function add(optional?: number, mandatory: string) {}
```

Both parameters of the previous function would be considered mandatory. Suppose that we call the function like so:

```
add(1);
```

The compiler would complain that you have not provided a value for the `mandatory` parameter. Remember, optional arguments are great, but place them last.

Default parameters

TypeScript gives us another feature to cope with default parameters, where we can set a default value that the parameter assumes when it's not explicitly passed upon executing the function. The syntax is pretty straightforward, as we can see when we refactor the previous example:

```
function greetMe(name: string, greeting: string = 'Hello'): string {
    return '${greeting}, ${name}';
}
```

Like optional parameters, default parameters must be put right after the required parameters in the function signature.

Rest parameters

One of the significant advantages of JavaScript flexibility when defining functions is the ability to accept an unlimited non-declared array of parameters. TypeScript can achieve the same behavior using the **rest** syntax. Essentially, we can define an additional parameter at the end of the arguments list prefixed by an ellipsis (…) and typed as an array:

```
function greetPeople(greeting: string, ...names: string[]): string {
    return greeting + ', ' + names.join(' and ') + '!';
}
```

Rest parameters are beneficial when we don't know how many arguments will be passed in.

Function overloading

Method and function overloading are typical in other languages, such as C#. However, implementing this functionality in TypeScript clashes with the fact that JavaScript, which TypeScript is meant to compile to, does not implement any elegant way to integrate it out of the box. So, the only workaround possible requires writing function declarations for each of the overloads and then writing a general-purpose function that wraps the actual implementation, whose list of typed arguments and return types are compatible with all the others:

```
function hello(names: string): string
function hello(names: string[]): string
function hello(names: any, greeting?: string): string {
    let namesArray: string[];
    if (Array.isArray(names)) {
        namesArray = names;
    } else {
        namesArray = [names];
    }
    if (!greeting) {
        greeting = 'Hello';
    }
    return greeting + ', ' + namesArray.join(' and ') + '!';
}
```

In the preceding example, we create three different function signatures, each of which features different type annotations. We could even define different return types by annotating the wrapping function with type any.

Arrow functions

ES6 introduced the concept of fat arrow functions (also called lambda functions in other languages such as Python, C#, Java, or C++). The purpose of the arrow function is to simplify the general function syntax and provide a bulletproof way to handle the function scope traditionally handled by the `this` keyword. The first thing we notice is its minimalistic syntax, where, most of the time, we see arrow functions as single-line, anonymous expressions:

```
const double = x => x * 2;
```

The preceding function computes the double of a given number x and returns the result, although we do not see any function or return statements in the expression. If the function signature contains more than one argument, we need to wrap them all between parentheses:

```
const add = (x, y) => x + y;
```

Arrow functions can also contain statements by wrapping the whole implementation in curly braces:

```
const addAndDouble = (x, y) => {
    const sum = x + y;
    return sum * 2;
}
```

Still, what does this have to do with scope handling? The value of `this` can point to a different context, depending on where we execute the function. When we refer to `this` inside a callback, we lose track of the upper context, which usually leads us to use conventions such as assigning its value to a variable named **self** or **that**. It is this variable that is used later on within the callback. Statements containing interval or timeout functions make for a perfect example of this:

```
function delayedGreeting(name): void {
    this.name = name;
    this.greet = function(){
        setTimeout(function() {
            console.log('Hello ' + this.name);
        }, 0);
    }
```

```
}
const greeting = new delayedGreeting('John');
greeting.greet();
```

If we execute the preceding script, it won't print the name `John` in the browser console as expected because it modifies the scope of `this` when evaluating the function inside the timeout call. If we modify the code according to the following, it will do the trick:

```
function delayedGreeting(name): void {
    this.name = name;
    this.greet = function() {
        setTimeout(() =>
            console.log('Hello ' + this.name)
        , 0);
    }
}
```

Even if we break down the statement contained in the arrow function into several lines of code wrapped by curly braces, the scope of `this` keeps pointing to the instance itself outside the timeout call, allowing for more elegant and clean syntax.

Now that we have acquired basic knowledge of functions in TypeScript and how they are executed, we can continue our journey to the typing system and learn some of the most common TypeScript features used in Angular.

Common TypeScript features

TypeScript has some general features that don't apply to classes, functions, or parameters but make coding more efficient and fun. The idea is that the fewer lines of code we write, the better it is. It's not only about fewer lines but also about making things more straightforward. There are many such features in ES6 that TypeScript has also implemented. In the following sections, we'll name a few that you will likely use in an Angular project.

Spread parameter

A **spread** parameter uses the same ellipsis syntax as the rest parameter but is used inside the body of a function. Let's illustrate this with an example:

```
const newItem = 3;
const oldArray = [1, 2];
const newArray = [...oldArray, newItem];
```

In the preceding snippet, we add an item to an existing array without changing the old one. The old array still contains `1`, `2`, whereas the new array contains `1`, `2`, and `3`. The current behavior is called immutability, which means not changing the old array but rather creating a new state from it. It is a principle used in functional programming as a paradigm and for performance reasons.

We can also use a spread parameter on objects:

```
const oldPerson = { name: 'John' };
const newPerson = { ...oldPerson, age: 20 };
```

In the preceding snippet, we are creating a merge between the two objects. Like in the array example, we don't change the previous variable, `oldPerson`. Instead, the `newPerson` variable takes the information from the `oldPerson` variable and adds its new values to it.

Template strings

Template strings are all about making your code clearer. Consider the following:

```
const url = 'http://path_to_domain' +
    'path_to_resource' +
    '?param=' + parameter +
    '&param2=' + parameter2;
```

So, what's wrong with the previous snippet? The answer is *readability*. It's hard to imagine what the resulting string will look like but editing the code by mistake and producing an unwanted result is also easy. To overcome this, we can use template strings in the following way:

```
const url =
'${baseUrl}/${path_to_resource}?param=${parameter}&param2={parameter2}';
```

The preceding syntax is a much more condensed expression and much easier to read.

Generics

Generics are expression indicating a general code behavior that we can employ, regardless of the data type. They 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:

```
function method<T>(arg: T): T {
    return arg;
}
method<number>(1);
```

In the preceding example, the type of `T` is not evaluated until we use the method. 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:

```
method<string>(1);
```

We specify that `T` should be a `string`, but we insist on passing it a value as a number. The compiler clearly states that this is not correct. You can, however, be more specific on what `T` should be. You can make sure that it is an array type so that any value you pass must adhere to this:

```
function method<T>(arg: T[]): T[] {
    console.log(arg.length);
    return arg;
}
class CustomPerson extends Array {}
class Person {}
const people: Person[] = [];
const newPerson = new CustomPerson();
method<Person>(people);
method<CustomPerson>(newPerson);
```

In the preceding case, we decide that `T` should be of `Person` or `CustomPerson` type and that the parameter needs to be of the array type. If we try to pass a single object, the compiler will complain:

```
const person = new Person();
method<Person>(person);
```

Alternatively, we can define that `T` should adhere to an interface like this:

```
interface Shape {
    area(): number;
}
class Square implements Shape {
    area() { return 1; }
}
class Circle implements Shape {
    area() { return 2; }
}
function allAreas<T extends Shape>(...args: T[]): number {
    let total = 0;
    args.forEach (x => {
```

```
        total += x.area();
    });
    return total;
}
allAreas(new Square(), new Circle());
```

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.

Optional chaining

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

```
const square = new Square();
```

In the preceding snippet, we create a `square` object using the `Square` class of the previous section. Later, we read the value of the `area` method by making sure that the object has a value set before reading it:

```
if (square !== undefined) {
    const area = square.area();
}
```

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

```
const area = square?.area();
```

The character `?` after the `square` object ensures that the `area` method will be accessed only if the object has a value. The case where optional chaining shines is in more complicated scenarios with much more values to check, such as the following:

```
const width = square?.area()?.width;
```

In the preceding scenario, we assume that the `area` property is an optional object that contains a `width` property. In that case, we would need to check values for both `square` and `area`.

Although the optional chaining feature was added in an earlier version of TypeScript, it has become very popular in the latest versions of Angular with its support in component templates.

Nullish coalescing

The nullish coalescing feature in TypeScript looks similar to the optional chaining we learned about in the previous section. However, it is more related to providing a default value when a variable is not set. Consider the following example that assigns a value to the `mySquare` variable only if the `square` object exists:

```
const mySquare = square ? square : new Square();
```

The previous statement is called a **ternary operator** and operates like a conditional statement. If the `square` object is **undefined** or **null**, the `mySquare` variable will take the default value of a new square object. We can rewrite the previous expression using nullish coalescing:

```
const mySquare = square ?? new Square();
```

Although the nullish coalescing feature was added in an earlier version of TypeScript, it has become very popular in the latest versions of Angular with its support in component templates.

Classes, interfaces, and inheritance

We have already overviewed the most relevant bits and pieces of TypeScript, and now it's time to see how everything falls into place with TypeScript classes. Classes are a fundamental concept in Angular development because everything in the Angular world is a TypeScript class.

Although the `class` is a reserved keyword in JavaScript, the language itself never had an actual implementation as in other languages such as Java or C#. JavaScript developers used to mimic this kind of functionality by leveraging the function object as a constructor type and instantiating it with the `new` operator. Other standard practices, such as extending function objects, were implemented by applying prototypal inheritance or using composition.

The class functionality in TypeScript is flexible and powerful enough to use in our applications. We already had the chance to tap into classes in the previous chapter. We'll look at them in more detail now.

Anatomy of a class

Property members are declared first in a class, then a constructor, and several other methods and property accessors follow. None contain the reserved `function` keyword, and all the members and methods are annotated with a type, except the constructor.

The following code snippet illustrates what a class looks like:

```
class Car {
    private distanceRun: number = 0;
    private color: string;

    constructor(private isHybrid: boolean, color: string = 'red') {
        this.color = color;
    }

    getGasConsumption(): string {
        return this.isHybrid ? 'Very low' : 'Too high!';
    }

    drive(distance: number): void {
        this.distanceRun += distance;
    }

    static honk(): string {
        return 'HOOONK!';
    }

    get distance(): number {
        return this.distanceRun;
    }
}
```

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

- **Members**: Any instance of the `Car` class will contain three properties: `color` typed as a `string`, `distanceRun` typed as a `number`, and `isHybrid` as a `boolean`. Class members will only be accessible from within the class itself. If we instantiate this class, `distanceRun`, or any other member or method marked as `private`, won't be publicly exposed as part of the object API.
- **Constructor**: The `constructor` `parameter` is executed when we create an instance of the class. Usually, we want to initialize the class members inside it with the data provided in the `constructor` signature. We can also leverage the signature to declare class members, as we did with the `isHybrid` property.

To do so, we need to prefix the `constructor` parameter with an access modifier such as `private` or `public`. As we learned when analyzing functions, we can define rest, optional, or default parameters, as depicted in the previous example with the `color` argument, which falls back to `red` when it is not explicitly defined.

- **Methods**: A method is a particular member representing a function and may return a typed value. It is a function that becomes part of the object API but can also be private. In this case, it can be used as a helper function within the internal scope of the class to achieve the functionalities required by other class members.
- **Static members**: Members marked as `static` are associated with the class and not with the object instances of that class. We can consume static members directly without having to instantiate an object first. Static members are not accessible from the object instances, which means they cannot access other class members using the `this` keyword. These members are usually included in the class definition as helper or factory methods to provide a generic functionality unrelated to any specific object instance.
- **Property accessors**: A property accessor is defined by prefixing a typed method with the name of the property we want to expose using the `set` (to make it writable) and `get` (to make it readable) keywords.

Constructor parameters with accessors

Typically, when we create a class, we give it a name, define a constructor, and create one or more fields, like so:

```
class Car {
    make: string;
    model: string;

    constructor(make: string, model: string) {
        this.make = make;
        this.model = model;
    }
}
```

For every field of the class, we usually need to do the following:

1. Add an entry to the constructor
2. Assign a value within the constructor
3. Declare the field

TypeScript eliminates the preceding boilerplate steps by using accessors on the `constructor` parameters:

```
class Car {
    constructor(public make: string, public model: string) {}
}
```

TypeScript will create the respective `public` fields and make the assignment automatically for us. As you can see, more than half of the code disappears; this is a selling point for TypeScript as it saves you from typing quite a lot of tedious code.

Interfaces

As applications scale and more classes are created, we need to find ways to ensure consistency and rule compliance in our code. One of the best ways to address the consistency and validation of types is to create interfaces. An **interface** is a code contract that defines a particular schema. Any artifacts such as classes and functions that implement 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.

In the following snippet, we're defining the `Vehicle` interface:

```
interface Vehicle {
    make: string;
}
```

Any class that implements the preceding interface must contain a member named `make`, which must be typed as a `string`:

```
class Car implements Vehicle {
    make: string;
}
```

Interfaces are also beneficial for defining the minimum set of members any artifact must fulfill, becoming an invaluable method to ensure consistency throughout our code base.

It is important to note that interfaces are not used just to define minimum class schemas but any type out there. This way, we can harness the power of interfaces by enforcing the existence of specific fields that are used later on as function parameters, function types, types contained in specific arrays, and even variables.

An interface may contain optional members as well. The following is an example of defining an interface that contains a required `message` and an optional `id` property member:

```
interface Exception {
    message: string;
    id?: number;
}
```

In the following snippet, we define the contract for our future class with a typed array and a method, with its returning type defined as well:

```
interface ErrorHandler {
    exceptions: Exception[];
    logException(message: string, id?: number): void
}
```

We can also define interfaces for standalone object types, which is quite useful when we need to define templated constructors or method signatures:

```
interface ExceptionHandlerSettings {
    logAllExceptions: boolean;
}
```

Let's bring them all together by creating a custom error handler class:

```
class CustomErrorHandler implements ErrorHandler {
    exceptions: Exception[] = [];
    logAllExceptions: boolean;

    constructor(settings: ExceptionHandlerSettings) {
        this.logAllExceptions = settings.logAllExceptions;
    }

    logException(message: string, id?: number): void {
        this.exceptions.push({message, id });
    }
}
```

The preceding class manages an internal array of exceptions. It also exposes the `logException` method to log new exceptions by saving them into an array. These two elements are defined in the `ErrorHandler` interface and are mandatory.

So far, we have seen interfaces as they are used in other high-level languages, but interfaces in TypeScript are stronger and more flexible; let's exemplify that. In the following code, we're declaring an interface, but we're also telling the TypeScript compiler to treat the `instance` variable as an `A` interface:

```
interface A {
    a: number;
}
const instance = { a: 3 } as A;
instance.a = 5;
```

An example of demonstrating the preceding code is to create a mocking library. When writing code, we might think about interfaces before we even start thinking about concrete classes because we know what methods need, but we might not have decided what methods will contain.

Imagine that you are building an order module. You have logic in your order module, and you know that, at some point, you will need to talk to a database service. You come up with an interface for the database service, and you defer the implementation of this interface until later. At this point, a mocking library can help you create a mock instance from the interface. Your code, at this point, might look something like this:

```
interface DatabaseService {
    save(order: Order): void
}
class Order {}
class OrderProcessor {

    constructor(private databaseService: DatabaseService) {}

    process(order) {
        this.databaseService.save(order);
    }

}

let orderProcessor = new OrderProcessor(mockLibrary.
mock<DatabaseService>());
orderProcessor.process(new Order());
```

Mocking at this point allows us to defer the implementation of `DatabaseService` until we are done writing the `OrderProcessor`. It also makes the testing experience a lot better. While in other languages, we need to bring in a mock library as a dependency, in TypeScript, we can utilize a built-in construct by typing the following:

```
const databaseServiceInstance = {} as DatabaseService;
```

In the preceding snippet, we create an empty object as a `DatabaseService`. However, be aware that you are responsible for adding a process method to your instance because it starts as an empty object. It will not raise any problems with the compiler; it is a powerful feature, but it is up to us to verify that what we create is correct. Let's emphasize how significant this TypeScript feature is by looking at some more cases where it pays off to be able to mock things.

Let's reiterate that the reason for mocking anything in your code is to make it easier to test. Let's assume your code looks something like this:

```
class Auth {
    srv: AuthService = new AuthService();

    execute() {
        if (srv.isAuthenticated()) {}
        else {}
    }
}
```

A better way to test this is to make sure that the `Auth` class relies on abstractions, which means that the `AuthService` should be created elsewhere and that we use an interface rather than a concrete implementation. So, we should modify our code so that it looks like this:

```
interface AuthService {
    isAuthenticated(): boolean;
}
class Auth {
    constructor(private srv: AuthService) {}

    execute() {
        if (this.srv.isAuthenticated()) {}
        else {}
    }
}
```

To test the preceding class, we would typically need to create a concrete implementation of the `AuthService` and use that as a parameter in the `Auth` instance:

```
class MockAuthService implements AuthService {
    isAuthenticated() { return true; }
}

const srv = new MockAuthService();
const auth = new Auth(srv);
```

It would, however, become quite tedious to write a mock version of every dependency that you wanted to mock. Therefore, mocking frameworks exist in most languages. The idea is to give the mocking framework an interface from which it would create a concrete object. You would never have to create a mock class, as we did previously, but that would be something that would be up to the mocking framework to do internally.

Class inheritance

Just like an interface can define a class, it can also extend the members and functionality of other classes. We can make a class inherit from another by appending the `extends` keyword to the class name, including the name of the class we want to inherit its members from:

```
class Sedan extends Car {
    model: string;

    constructor(make: string, model: string) {
        super(make);
        this.model = model;
    }
}
```

In the preceding class, we extend from a parent `Car` class, which already exposes a member called `make`. We can populate the members by the parent class and execute their constructor using the `super` method, which points to the parent constructor. We can also override methods from the parent class by appending a method with the same name. Nevertheless, we can still execute the original parent's class methods as it is still accessible from the `super` object.

Classes and interfaces are basic features of the TypeScript language. As we will see in the following section, decorators enhance the use of classes in an application by extending them with custom functionality.

Decorators

Decorators are a very cool functionality that allows us to add metadata to class declarations for further use. By creating decorators, we are defining special annotations that may impact how our classes, methods, or functions behave or simply altering the data we define in fields or parameters. They are a powerful way to augment our type's native functionalities without creating subclasses or inheriting from other types. It is, by far, one of the most exciting features of TypeScript. It is extensively used in Angular when designing modules and components or managing dependency injection, as we will learn later in *Chapter 6*, *Managing Complex Tasks with Services*.

The `@` prefix recognizes decorators in their name, which are standalone statements above the element they decorate. We can define up to four different types of decorators, depending on what element each type is meant to decorate:

- Class decorators
- Property decorators
- Method decorators
- Parameter decorators

The Angular framework defines custom decorators, which we will use during the development of an application.

We'll look at the previous types of decorators in the following subsections.

Class decorators

Class decorators allow us to augment a class or perform operations on its members. The decorator statement is executed before the class gets instantiated. Creating a class decorator requires defining a plain function, whose signature is a pointer to the constructor belonging to the class we want to decorate. The formal declaration defines a class decorator as follows:

```
declare type ClassDecorator = <TFunction extends
Function>(Target:TFunction) => TFunction | void;
```

Let's see how we can use a class decorator through a simple example:

```
function Banana(target: Function): void {
    target.prototype.banana = function(): void {
        console.log('We have bananas!');
```

```
    }
}
@Banana
class FruitBasket {}

const basket = new FruitBasket();
basket.banana();
```

In the preceding snippet, we use the `banana` method, which was not initially defined in the `FruitBasket` class. However, we decorate it with the `@Banana` decorator. It is worth mentioning, though, that this won't compile. The compiler will complain that `FruitBasket` does not have a `banana` method, and rightfully so because TypeScript is typed. So, at this point, we need to tell the compiler that this is valid. So, how do we do that? One way is that, when we create our `basket` instance, we give it the type `any`:

```
const basket: any = new FruitBasket();
```

The compiler will not complain about the method now, and the compilation of our code will complete successfully.

Extending a class decorator

Sometimes, we might need to customize how a decorator operates upon instantiating it. We can design our decorators with custom signatures and then have them return a function with the same signature we defined without parameters. The following piece of code illustrates the same functionality as the previous example, but it allows us to customize the message:

```
function Banana(message: string) {
    return function(target: Function) {
        target.prototype.banana = function(): void {
            console.log(message);
        }
    }
}
@Banana('Bananas are yellow!')
class FruitBasket {}

const basket: any = new FruitBasket();
basket.banana();
```

If we run the preceding code, the browser console will print the following message:

```
Bananas are yellow!
```

As a rule of thumb, decorators that accept parameters require a function whose signature matches the parameters we want to configure. The function also returns another function that matches the signature of the decorator.

Property decorators

Property decorators are applied to class fields and are defined by creating a function whose signature takes two parameters:

- `target`: The prototype of the class we want to decorate
- `key`: The name of the property we want to decorate

Possible use cases for this decorator are logging the values assigned to class fields when instantiating objects or reacting to data changes in such fields. Let's see an actual example that showcases both behaviors:

```
function Jedi(target: Object, key: string) {
    let propertyValue: string = target[key];
    if (delete target[key]) {
        Object.defineProperty(target, key, {
            get: function() {
                return propertyValue;
            },
            set: function(newValue){
                propertyValue = newValue;
                console.log('${propertyValue} is a Jedi');
            }
        });
    }
}
class Character {
    @Jedi
    name: string;
}
const character = new Character();
character.name = 'Luke';
```

The preceding snippet follows the same logic as for parameterized class decorators. However, the signature of the returned function is slightly different to match that of the parameterless decorator declaration we saw earlier.

Let's now see an example that depicts how we can log changes on a given class property using a property decorator:

```
function NameChanger(callbackObject: any): Function {
    return function(target: Object, key: string): void {
        let propertyValue: string = target[key];
        if (delete target[key]) {
            Object.defineProperty(target, key, {
                get: function() {
                    return propertyValue;
                },
                set: function(newValue) {
                    propertyValue = newValue;
                    callbackObject.changeName.call(this, propertyValue);
                }
            });
        }
    }
}
```

The `NameChanger` decorator can be applied in a class to be executed when the `name` property is modified:

```
class Character {
    @NameChanger ({
        changeName: function(newValue: string): void {
            console.log('You are now known as ${newValue}');
        }
    })
    name: string;
}
var character = new Character();
character.name = 'Anakin';
```

In the preceding snippet, the `changeName` function is triggered when the value of the property changes in the `character` instance.

Method decorators

A method decorator can detect, log, and intervene in how methods are executed. To do so, we need to define a function whose payload takes the following parameters:

- `target`: Represents the decorated method.
- `key`: The actual name of the decorated method.
- `descriptor`: A property descriptor of the given method. It is a hash object containing, among other things, a property named `value` that references the method itself.

In the following example, we're creating a decorator that displays how a method is called:

```
function Log(){
    return function(target, key: string, descriptor: PropertyDescriptor) {
        const oldMethod = descriptor.value;
        descriptor.value = function newFunc(...args:any[]){
            let result = oldMethod.apply(this, args);
            console.log('${key} is called with ${args.join(',')} and
result ${result}');
            return result;
        }
    }
}

class Hero {
    @Log()
    attack(...args:[]) { return args.join(); }
}

const hero = new Hero();
hero.attack();
```

The preceding snippet also illustrates what the arguments were upon calling the method and the result of the method's invocation.

Parameter decorator

A parameter decorator, the last one we will learn about, taps into parameters located in function signatures. It is not intended to alter the parameter information or the function behavior but to look into the parameter value and perform operations such as logging or replicating data. It accepts the following parameters:

- `target`: The object prototype where the function, whose parameters are decorated, usually belongs to a class
- `key`: The name of the function whose signature contains the decorated parameter
- `index`: The index where this decorator has been applied in the parameter's array

The following example shows a working example of a parameter decorator:

```
function Log(target: Function, key: string, index: number) {
    const functionLogged = key || target.prototype.constructor.name;
    console.log('The parameter in position ${index} at ${functionLogged}
has been decorated');
}
class Greeter {
    greeting: string;

    constructor (@Log phrase: string) {
        this.greeting = phrase;
    }
}
```

In the preceding snippet, we declare the `functionLogged` variable in that way because the value of the target parameter varies depending on the function whose parameters are decorated. Therefore, decorating a constructor or a method parameter is different. The former returns a reference to the class prototype, while the latter returns a reference to the `constructor` function. The same applies to the `key` parameter, which is undefined when decorating the `constructor` parameters.

Parameter decorators do not modify the value of the parameters decorated or alter the behavior of the methods or constructors where these parameters live. They usually log or prepare the object created to implement additional layers of abstraction or functionality through higher-level decorators, such as a method or class decorator. Common use case scenarios for this encompass logging component behavior or managing dependency injection.

Advanced types

We have already learned about some basic types in the TypeScript language that we usually meet in other high-level languages. In this section, we'll look at some advanced types that will help us during Angular development.

Partial

The `Partial` type is used when we want to create an object from an interface but include some of its properties:

```
interface Hero {
    name: string;
    power: number;
}
const hero: Partial<Hero> = {
    name: 'Boothstomper'
}
```

In the preceding snippet, we can see that the `hero` object does not include `power` in its properties.

Record

Some languages, such as C#, have a reserved type when defining a key-value pair object or dictionary, as it is known. In TypeScript, there is no such thing. If we want to define such a type, we declare it as follows:

```
interface Hero {
    powers: {
        [key: string]: number
    }
}
```

However, the preceding syntax is not clear. In a real-world scenario, interfaces have many more properties. Alternatively, we can use the `Record` type to define the interface:

```
interface Hero {
    powers: Record<string, number>
}
```

It defines the key as a `string`, which is the name of the power in this case, and the value, which is the actual power factor, as a `number`.

Union

We've already learned about generics and how they can help us when we want to mix types. A nice alternative, when we know what the possible types are, is the `Union` type:

```
interface Hero {
    name: string;
    powers: number[] | Record<string, number>;
}
```

In the preceding snippet, we define the `powers` property as an array of numbers or a key-value pair collection.

And that wraps up advanced types. As we learned, the TypeScript typing system is very flexible and allows us to combine types for more advanced scenarios.

In the following section, we will learn how to use modules with TypeScript.

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.

A module works at a file level, where each file is the module itself, and the module name matches the filename without the `.ts` extension. Each member marked with the `export` keyword becomes part of the module's public API. Consider the following module that is declared in a `my-service.ts` file:

```
export class MyService {
    getData() {}
}
```

To use the preceding module and its exported class, we need to import it into our application code:

```
import { MyService } from './my-service';
```

The `./my-service` path is relative to the location of the file that imports the module. If the module exports more than one artifact, we place them inside the curly braces one by one, separated with a comma:

```
export class MyService {
    getData() {}
```

```
}
export const PI = 3.14;
import { MyService, PI } from './my-service';
```

In the preceding example, the `MyService` class exports the `getData` method and the `PI` variable in one go.

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 gave us the chance 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. Modules and decorators were some other significant features we explored in this chapter. As we will see very soon, having sound knowledge of these mechanisms is paramount to understanding how dependency injection works in Angular.

With all this knowledge at our disposal, we can start learning how to apply it by building Angular applications. In the next chapter, we will learn how to use Angular modules, which are not the same as JavaScript modules, to structure an Angular application. We will see in detail what Angular modules are and how we can use them to organize an Angular application in an efficient and properly structured way.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`

3

Organizing Application into Modules

The purpose of a web application is to provide end users with a specific set of functionalities to fulfill their particular needs using a web interface. The functionality of the web application is organized into cohesive blocks of features.

Application features in an Angular application are represented using Angular modules. An Angular module groups specific functionality concerned with a particular application domain or user workflow.

In this chapter, we will cover the following topics:

- Introducing Angular modules
- Creating our first module
- Grouping application features into modules
- Leveraging Angular built-in modules

Technical requirements

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

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Introducing Angular modules

Angular modules are containers for a particular block of code that adheres to the same functionality. An Angular module is dedicated to an application domain, such as orders or customers for an e-shop application, or a user workflow, such as order checkout or user registration. Generally, it addresses a particular set of capabilities that an application can have.

The main advantage of the Angular module architecture is that it scales better and is easier to test. If we think of a module as a particular feature of an application, it allows us to organize our Angular application so that we can develop a particular piece of functionality independently of the others. It dramatically enhances team management in large organizations where each development team can work in a separate feature. Features can gradually be deployed, ensuring the seamless operation of our application.

Angular modules are different from JavaScript modules due to the context in which they operate. As we learned in the previous chapter, JavaScript modules help us organize our code in multiple files. On the other hand, Angular modules work in the context of the Angular framework. They play a significant role when the framework compiler converts an Angular application into JavaScript code. The role of an Angular module is to group all the artifacts of an Angular application that share common functionality.

As we learned in *Chapter 1*, *Building Your First Angular Application*, a typical Angular application contains at least a main module called `AppModule` that is defined in the `app.module.ts` file:

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

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

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

The `import` statements at the top of the file are used to import the following artifacts into the main Angular module:

- `NgModule`: This is used to configure an Angular module.
- `BrowserModule`: Configures an Angular application to run in the browser platform. We have already learned that the Angular framework is cross-platform in *Chapter 1*, *Building Your First Angular Application*. We would import a different module to run the Angular application on another platform, such as the server.
- `AppComponent`: The main component of the Angular application.

The `NgModule` is an Angular decorator similar to the TypeScript decorators we already saw in the previous chapter. It accepts an object as a parameter with the following properties:

- `declarations`: This contains Angular artifacts that share common functionality bound to a specific application feature. Artifacts that can be added to this property are Angular components, directives, and pipes, which we will see in more detail later in the book. The main module of the application contains the main component by default.
- `imports`: This contains other Angular modules whose declarations are needed by the current module. When an Angular module needs to use features from another module, it must import it first to start using it. The main application module imports `BrowserModule` because it needs its functionality for loading the current application into the browser.

The `imports` array should not be confused with the `import` statements at the top of the module file. The former is used to import functionality from other Angular modules into the current module, whereas the latter is for importing their respective JavaScript modules.

- `providers`: It contains special-purpose Angular artifacts that are called **services**. Services handle complex tasks in an Angular application, such as communicating with an HTTP endpoint or interacting with a browser API. We will learn more about them in *Chapter 6*, *Managing Complex Tasks with Services*. Initially, the main application module does not need any services. Services provided in the main application module are accessible application-wide.
- `bootstrap`: Defines the component that will be loaded at application startup. The `bootstrap` property is set only once in the main application module and is usually the main component. You should not change it unless there is a compelling reason.

The `AppModule` TypeScript class is empty because Angular modules usually do not contain any logic. As we have learned, the primary purpose of an Angular module is to group artifacts with similar functionality. Their purpose is fulfilled by using the `@NgModule` decorator above the class. The Angular framework would treat `AppModule` as a single TypeScript class if the decorator was missing. It actually tells Angular that this is *indeed* an Angular module.

In addition, to the main application module, we can create other Angular modules that represent features of the application and are typically called **feature modules**. In the next section, we will learn how to create feature modules using the Angular CLI.

Creating our first module

When creating a new Angular application, the first step is to define the different features our application needs. We should remember that each one should make sense on its own in isolation from the others. Once we've defined the required features, we will create a module for each. Each module will then be filled with the Angular artifacts that shape the feature it represents. Always remember the principles of encapsulation and reusability when defining your feature set.

If we wanted to create an e-shop application, we would typically need a module to manage the stock of our products. To create a new module in an Angular application, we use the `generate` command of the Angular CLI, passing the name of the module as a parameter:

```
ng generate module products
```

The preceding command will create a `products` folder inside the `src\app` folder of our Angular workspace. The `products` folder is the container for Angular artifacts that contain functionality related to the products module.

Keeping all module artifacts inside the module's folder is good practice. It helps you visualize the functionality of a module at a glance and concentrate on that specific module during development. It is also helpful when you decide to make a refactor to your code and must move the whole module.

It will also create the `products.module.ts` file, which is the TypeScript file that contains the Angular module:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
```

```
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class ProductsModule { }
```

The preceding module seems slightly different from the main application module we saw earlier because it currently contains less functionality.

The `declarations` array is initially empty because our module has no functionality yet. When we add new Angular artifacts to our module to implement specific functionality, they will appear in that array. The next chapter will teach us how to create a component in an Angular module.

The `imports` array contains the `CommonModule` by default. The `CommonModule` is a built-in module of the Angular framework that contains common Angular artifacts we usually would like to use in an Angular application.

When importing other modules, we should take extra precautions not to cause any cyclic dependencies by using one that has already imported our module. Otherwise, we may end up with circular reference issues.

We will learn more about other built-in modules of the Angular framework in the *Leveraging Angular built-in modules* section. In real-world applications, feature modules can also share their encapsulated functionality with other modules. We will learn how to accomplish that in the following section.

Group application features into modules

Each Angular module represents a particular feature of an Angular application. The way these feature modules are added to the application depends on a business's needs. In this section, we will investigate three different ways:

- Adding modules in the main application module
- Adding modules to another feature module
- Grouping feature modules by type

In the following sections, we will explore each way in more detail.

Add a module in the main module

We have already learned how to create a new Angular module for our application using the Angular CLI. Creating an Angular module does not make it automatically available to the application. It is our responsibility to register the new module with the rest of the application using the main application module:

1. Open the `app.module.ts` file and add a new `import` statement to import `ProductsModule`:

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

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

2. Add the `ProductsModule` class into the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ProductsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Our Angular application is now fully aware of the new module we created. Importing an Angular module into the main application module extends the capabilities of the application because it adds the existing functionality of the imported module into the Angular application.

The purpose of the main application module is to orchestrate the interaction of feature modules throughout the application and is not tied to a specific feature. Generally, we should never touch the `app.module.ts` file, except to import a new feature module to add new functionality to the application.

The following is a representation of an Angular application organized into three features:

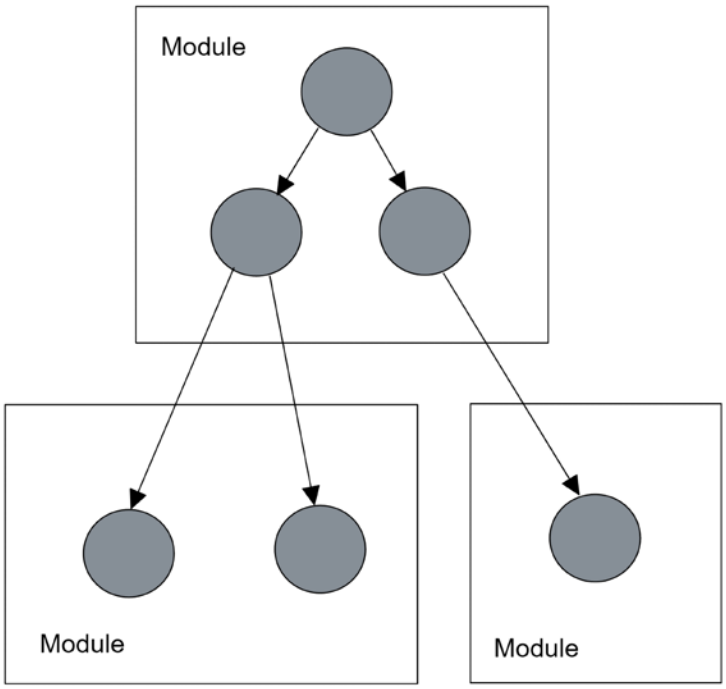


Figure 3.1: Module hierarchy

In the preceding diagram, each circle represents a particular block of functionality implemented using Angular components. We will learn more about components in *Chapter 4*, *Enabling User Experience with Components*.

Additional to the main application module, other feature modules may need existing functionality from the products module. In this case, we need to follow the same process as we did for the main module to make it available to other modules as well. In the following section, we will learn how to expose functionality from the products module to other feature modules of an application.

Exposing feature modules

An e-shop web application will generally need a module to manage orders. We will use the Angular CLI to generate a new Angular module called `orders`:

```
ng generate module orders
```

Customers must select products from a list when they place a new order through the application. The product list will be a particular block of functionality that exists in the products module. Consider that the product list is an existing component inside the products module called `ProductListComponent`.

We will not implement the `ProductListComponent` yet because components will be covered in the next chapter.

The TypeScript file of the products module would look like the following:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';

@NgModule({
  declarations: [
    ProductListComponent
  ],
  imports: [
    CommonModule
  ]
})
export class ProductsModule { }
```

As we have already learned, when a module needs functionality from another feature module, it must import it. So, the orders module must only add `ProductsModule` in its `imports` array:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductsModule } from '../products/products.module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    ProductsModule
  ]
```

```
})
export class OrdersModule { }
```

Remember to always add the `import` statement at the top of the file first; otherwise, Angular will not be able to recognize `ProductsModule`.

VS Code will also help by adding the `import` statement directly after typing the module name in the `imports` array. If this does not work, you can do it manually by clicking on `ProductsModule` and selecting the **Code Actions** menu from the blue bulb icon.

However, the preceding code is insufficient because Angular modules use a public API to communicate with other modules in an application. All components of an Angular module are *not* exposed to that API by default. We have already seen one part of the API: the `imports` array of the `@NgModule` decorator. To expose a component through the API, we need to use another part of the decorator: the `exports` array. When we want to make a component publicly available to other modules that import an existing feature module, we need to add it to the `exports` array of the module that owns it:

```
@NgModule({
  declarations: [
    ProductListComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [ProductListComponent]
})
```

In the preceding code, the orders module can now use the product list from the products module normally.

The `exports` array should not be confused with the `export` statement in front of the TypeScript class. The former is used for Angular modules, whereas the latter is to export the TypeScript class so that it can be imported into other Angular module files.

So far, we have seen two types of modules: the main application module and the feature module. There are also other types of modules that we can use in Angular applications that serve specific purposes and needs. We'll look at these in the next section.

Organizing modules by type

Angular modules are used to group similar functionality and provide this to other modules. They can be further organized by the type of functionality and how an Angular application loads them. We can separate modules according to the feature that they represent:

- **Core module**: This usually contains application-wide artifacts that do not fit in a specific module. Such artifacts are components that are loaded once in an application, such as a top bar that contains the main menu of the application, a footer component with copyright information, or a loading spinner. It also contains services that can be shared among modules, such as a local cache service or a custom logger. This module is usually called **core,** and the accompanying TypeScript file is named `core.module.ts`. The core module should be loaded only once in the main application module.
- **Shared module**: This contains Angular artifacts such as components, directives, and pipes that can be used in multiple feature modules. It may also provide a container for other exported modules that contain reusable artifacts. This module is usually called **shared,** and the accompanying TypeScript file is named `shared.module.ts`. The shared module is imported into each feature module that wants to use its exported artifacts.

The preceding list is a recommendation on how to group your Angular modules for specific types of artifacts. However, the flexibility of the Angular framework allows you to use a custom grouping that fits your particular business needs.

We can also distinguish between modules according to how the Angular framework loads them:

- **Eager-loaded modules**: These are loaded at the application startup. We can distinguish between an eagerly loaded module by whether it is declared in the `imports` array of another module or not. All feature modules we have used in the chapter are eagerly loaded.
- **Lazy-loaded modules**: These are loaded on-demand, usually due to in-app navigation. Lazy-loaded modules are not declared in the `imports` array of a module, but they have their specific way of loading, as we will learn in *Chapter 9*, *Navigate through Application with Routing*.

How Angular loads a module is directly related to the final bundle of an application. In *Chapter 13*, *Bringing Application to Production*, we will see that how we load our Angular modules affects the build process of our application directly.

We have already learned how to create Angular modules and add them to an Angular application. In the following section, we will see some of the built-in modules that the Angular framework provides us out of the box.

Leveraging Angular built-in modules

We have already learned that the Angular framework contains a rich collection of first-party libraries that can help us during the development of an Angular application. The functionality of each library is exposed through an Angular module. We need to import these modules into our applications to start using their functionality, as with any other module in Angular. Below are some of the most widely used modules of the Angular framework:

- `BrowserModule`: This is used to run Angular applications in the browser and must be imported only once in an Angular application.
- `CommonModule`: This contains specific Angular artifacts that support the Angular template syntax and enrich our HTML templates. Typical examples include directives for looping or displaying HTML content conditionally and applying CSS styles in HTML. We will work most of the time with this module in this book.
- `FormsModule`/`ReactiveFormsModule`: This allows us to build HTML forms for interacting with user input data. We will learn more about **Angular Forms** in *Chapter 10*, *Collecting User Data with Forms*.
- `HttpClientModule`: This enables communication and data exchange with a remote endpoint over HTTP. We will learn more about HTTP communication in *Chapter 8*, *Communicating with Data Services over HTTP*.
- `RouterModule`: This performs and handles navigation in an Angular application. We will learn more about **Angular Router** in *Chapter 9*, *Navigate through Application with Routing*.
- `BrowserAnimationsModule`: This cooperates with the **Angular Material** library and enables UI animations in an Angular application. We will learn more about Angular Material in *Chapter 11*, *Introduction to Angular Material*.

The preceding list contains Angular modules you will primarily use in an Angular application as you start out. The Angular framework contains a lot more for many business needs and use cases.

Summary

Angular modules are an essential part of an Angular application. They define the main features of the application and organize them cohesively and efficiently.

We learned the purpose of a module in an Angular application and how different they are from JavaScript modules. We also explored the structure of an Angular module and how Angular uses decorators to configure it.

We also learned the different ways to add a module in an Angular application and how modules communicate with each other over a public API. Finally, we saw how we could group modules according to their features and some of the most widely used Angular built-in modules.

In the next chapter, we will learn more about the functionality that goes into an Angular module and how it can be represented as an Angular component.

4

Enabling User Experience with Components

So far, we have had the opportunity to take a bird's-eye overview of the Angular framework. We learned how to create a new Angular application using the Angular CLI and how to group application features into modules. We saw how to use Angular modules and organize our application in an efficient and cohesive way that empowers scalability and testability.

As we learned, Angular modules extend our Angular applications by adding extra functionality. We have already mentioned that the functionality of a module is mainly represented using Angular components. We seem to have everything we need to explore 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

The chapter contains various code samples to walk you through the concept of Angular components. You can find the related source code in the `ch04` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Creating our first component

Components are the basic building blocks of an Angular application. They control different parts of a web page 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:

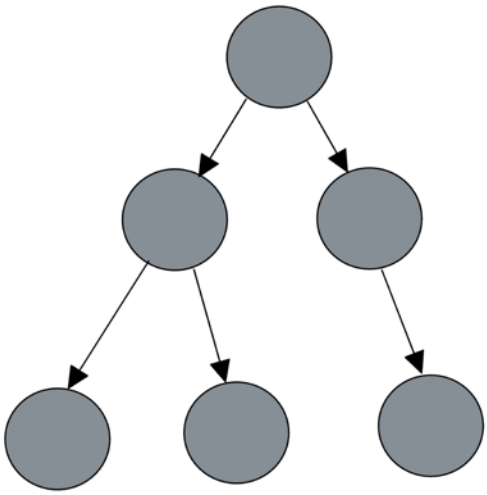


Figure 4.1: Component architecture

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 in the previous diagram, 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
- Registering components with modules
- Creating **standalone** components

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

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 file of the component is defined in the `app.component.ts` file:

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

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

The `import` statement at the top of the file is used to import the `Component` artifact from the `@angular/core` `npm` package. The `Component` artifact is an Angular decorator that is used to configure an Angular component. It contains 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.
- `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.
- `styleUrls`: Defines a list of paths that point to external CSS style sheet files. Alternatively, you can provide the styles inline using the `styles` property.

The TypeScript file of the component also has a TypeScript class called `AppComponent` that contains a `title` property. The `@Component` decorator above the class tells Angular that it is an Angular component. If the decorator was missing, the Angular framework would treat it as a simple TypeScript class.

Registering components with modules

In addition to the main application component, we can create other Angular components that provide specific functionality to the Angular module. In the previous chapter, we created an Angular module for managing the products of our e-shop application and used an imaginary product list component. Now it is time to create this component for real!

To create a new component in an Angular application, we use the `generate` command of the Angular CLI, passing the name of the component as a parameter. Run the following command inside the `products` folder we created in the previous chapter:

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

The preceding command will create the `product-list` component and register it with the products module.

If we run the `generate` command in the `src\app` folder, it will register it with the main application module. We do not want this because it violates the principle of modularity and re-usability of Angular modules.

Creating an Angular component is a two-step process. It includes creating the necessary files of the component and registering it with an Angular module. The preceding command will create a `product-list` folder that contains the individual component files we learned about in *Chapter 1, Building Your First Angular Application*. At the same time, the Angular CLI will register the specific component with the `products` module by adding the `ProductListComponent` class in the `declarations` array of the `products.module.ts` file:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';

@NgModule({
  declarations: [
    ProductListComponent
  ],
  imports: [
    CommonModule
  ]
```

```
})
export class ProductsModule { }
```

Angular components must be registered with only one Angular module.

When we register a component with an Angular module, we give it a **compilation context**. The component can find everything that needs to be loaded inside that context. However, we can create components that do not exist in the context of a specific Angular module.

Creating standalone components

A component that is not registered with an Angular module is called a standalone component. Standalone components do not need a compilation context from an Angular module because they import any Angular artifacts they need by themselves. To create a standalone component using the Angular CLI, we pass the `standalone` option in the `generate` command that we learned about earlier:

```
ng generate component product --standalone
```

The TypeScript file of a standalone component is slightly different, as we can see in the `product.component.ts` file:

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

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

The `@Component` decorator contains the following additional properties:

- `standalone`: Indicates whether a component is standalone or not.
- `imports`: Contains Angular modules or other *standalone* components that the component needs to be loaded correctly. The Angular CLI adds `CommonModule` by default when generating new standalone components.

Looking closely, you may notice that the `@Component` decorator does the same job as an Angular module regarding imported artifacts. It looks like we moved the `imports` array from the Angular module to the component decorator. Also, you will notice that the component generation did not modify any Angular module.

Standalone components can import Angular modules and vice versa. All we have to do is add the standalone component in the `imports` array of the module as if it was a module itself:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';
import { ProductComponent } from './product/product.component';

@NgModule({
  declarations: [
    ProductListComponent
  ],
  imports: [
    CommonModule,
    ProductComponent
  ]
})
export class ProductsModule { }
```

Importing `ProductComponent` into the `ProductsModule` makes the standalone component available module-wide.

A standalone component should not be added to the `declarations` array of an Angular module because that would make it registered with that module.

Standalone components are a revolutionary way to adopt a simpler and component-centric approach to building Angular applications. We learned how to create them, and we also saw how to create components and register them with an Angular module.

Standalone components are recommended for quick prototyping, demo purposes, or when learning Angular for the first time. As you progress and add more features to your application, you may need to use Angular modules to organize your component hierarchy better.

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 for interacting 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 component template land by exploring how we render the component on the web page.

Loading the component template

We have already learned that Angular uses the selector 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 need to 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 of the Angular application we are currently working on and replace its content with the following:

```
<app-product-list></app-product-list>
```

After you add the preceding snippet, you will notice that you get the following error in the editor:

Figure 4.2: Template error

The previous error is caused because `ProductsModule` does not yet expose the product list component through its public API.

2. Open the `products.module.ts` file and add the `ProductListComponent` class in the `exports` array of the @NgModule decorator:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';

@NgModule({
  declarations: [
    ProductListComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [ProductListComponent]
```

```
})
export class ProductsModule { }
```

3. Run the following command to start the Angular application:

```
ng serve
```

After the application has been built successfully, navigate to `http://localhost:4200` to preview it. The web page displays the following text:

```
product-list works!
```

The displayed text is the content of the component template that exists inside the `product-list.component.html` file. The Angular CLI creates a default HTML template when creating a new component that consists of an HTML paragraph element containing the component selector:

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

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 from the component class to the template:

```
<span>{{title}}</span>
```

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 `span` HTML element, a method called **property binding**:

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

In the preceding snippet, we bind to the **Document Object Model** (**DOM**) property of an element, not an 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 in a browser, it parses the HTML content of the page and converts it into a tree structure, the DOM. Each HTML element of the page is converted to 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 need first to understand how Angular interacts with attributes and properties. It defines attributes in HTML to initialize a DOM property, 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. 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 in the corresponding Angular component. Remember that property binding interacts with the properties of Angular components. 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:

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

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

Property binding in the Angular framework is convenient for displaying data and styling purposes.

Styling the component

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

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

The Angular framework provides two types of property binding to set both of them dynamically, **class binding** and **style 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 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, evaluates to `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 to them. 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 a semicolon 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.

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**. Consider the following HTML snippet:

```
<button (click)="onClick()">Click me</button>
```

An event binding listens for DOM events on the target HTML element and responds to those events by calling corresponding methods in the component class. In the preceding case, the component calls the `onClick` method when the user clicks the button. The event inside parentheses is called the **target event** and is the event we are currently listening to. Event binding in Angular supports all native DOM events found at `https://developer.mozilla.org/docs/Web/Events`.

The expression on the right is called the **template statement** and corresponds to the `onClick` method of the component class.

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

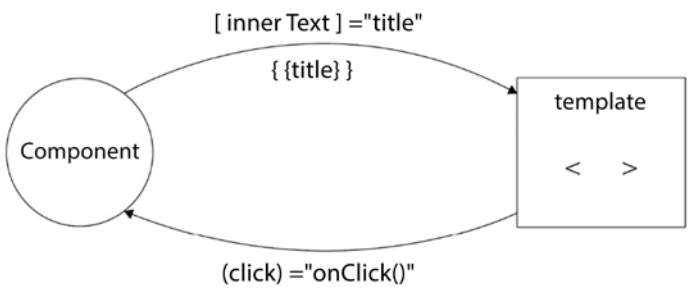


Figure 4.3: Component-template interaction

The same principle we follow for interacting with the component template and class can be used when we want to communicate between components.

Component inter-communication

In a nutshell, 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 state of the component.

Let's look at how Angular solves the problem of injecting data into and extracting data from components through quick and easy examples in the following sections.

Passing data using an input binding

We will expand our `products` module and create a new component that will display the details of a product, such as a name and a price. Data representing the specific product details will be dynamically passed from the product list component.

For now, we will only pass and display the name of the product. To follow along with code samples, copy the CSS styles from the `styles.css` file of the GitHub repository, as defined in the *Technical requirements* section.

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

1. Run the following Angular CLI command inside the `src\app\products` folder of the project to create the new Angular component:

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

2. Open the `product-detail.component.ts` file of the new component and import the `Input` artifact from the `@angular/core` npm package:

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

   The `Input` artifact is an Angular property decorator that is used when we want to pass data from one component *down* to another component.

3. Define a `name` property in the `ProductDetailComponent` class that uses the `Input` decorator and initialize it to an empty string:

```
@Input() name = '';
```

4. Now, open the `product-detail.component.html` file and add the following code to display the name of a product:

```
<h2>Product Details</h2>
<h3>{{name}}</h3>
```

   In the preceding template, we use the Angular interpolation syntax to convert the `name` property to text and display it on the page.

We have already completed most of the work; we now need to pass the value of the `name` input property from the product list component so that it can be adequately displayed in the product detail component:

1. Open the `product-list.component.ts` file and create a `selectedProduct` property in the `ProductListComponent` class:

```
selectedProduct = '';
```

2. Open the `product-list.component.html` file and replace its content with the following HTML template:

```
<h2>Product List</h2>
<ul>
  <li (click)="selectedProduct = 'Webcam'">Webcam</li>
  <li (click)="selectedProduct = 'Microphone'">Microphone</li>
  <li (click)="selectedProduct = 'Wireless Keyboard'">Wireless
Keyboard</li>
</ul>
```

   In the preceding code, we have created a list of products using an unordered list HTML element. When the user clicks on a product, the `selectedProduct` property is set accordingly through the binding in the `click` event of the `<li>` element.

3. Now, add the following snippet after the list of products to load the product details component:

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

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

There are cases where we want to pass a static text or a value that we are sure will never change. In these cases, we can omit the square brackets of the input binding as follows:

```
<app-product-detail name="Webcam"></app-product-detail>
```

If we now run the application, we should see the following product list:

Figure 4.4: Product list

Notice that the product details section is displayed, although we have not selected a product yet. We will learn how to solve that in the following chapter using Angular built-in directives.

Clicking on a product from the list should display the product name under the **Product Details** section:

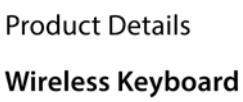


Figure 4.5: Product details

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 around, from the child component to the parent? How do we notify the parent component about specific actions that occur in the child component?

Consider a scenario where the product details 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` and `EventEmitter` artifacts from the `@angular/core` npm package:

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

   The `Output` artifact is an Angular property decorator that is used when we want to create events that will be triggered from one component *up* to another. The `EventEmitter` class is used to emit those events.

2. Define a new component property that uses the `Output` decorator and is initialized to a new `EventEmitter` object:

```
@Output() bought = new EventEmitter();
```

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

```
buy() {
  this.bought.emit();
}
```

   The `buy` method calls the `emit` method on the `bought` output event we created in the previous step. The `emit` method emits an event and notifies any component currently listening to the event.

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

```
<h2>Product Details</h2>
<h3>{{name}}</h3>
<button (click)="buy()">Buy Now</button>
```

5. 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:

```
onBuy() {
  window.alert('You just bought ${this.selectedProduct}!');
}
```

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

6. Finally, modify the selector of the product detail component in the `product-list.component.html` file:

```
<app-product-detail [name]="selectedProduct" (bought)="onBuy()"></
app-product-detail>
```

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

Select a product from the list and click on the **Buy Now** button in the currently running application. You should see something like the following:

Figure 4.6: Alert window

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

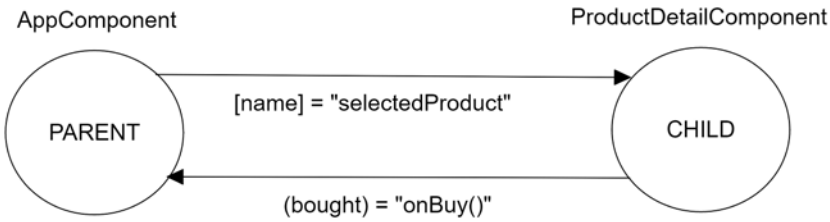


Figure 4.7: Component inter-communication

The output event of the product details 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 `EventEmitter` property can accept any data to pass up to the parent component. The proper way is initially to define the data type that can be passed to the `EventEmitter` property.

Currently, the product list component already knows the selected product. So, it already knows the product that the user bought. Let's assume this was not the case, and the product list component could only know the selected product after the user clicks on the **Buy Now** button. We would use generics of the `EventEmitter` class to declare the type of data that would be passed into the product list component:

```
@Output() bought = new EventEmitter<string>();
```

The `buy` method of the `ProductDetailComponent` class would then call the `emit` method, passing a string value:

```
buy() {
  this.bought.emit(this.name);
}
```

The data would be available in the template of the product list component through an `$event` object:

```
<app-product-detail [name]="selectedProduct" (bought)="onBuy($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. Additionally, the signature of the `onBuy` method in the `ProductListComponent` class should change accordingly:

```
onBuy(name: string) {
  window.alert('You just bought ${name}!');
}
```

Input and output bindings are a great way to communicate between components using the public API. There are cases, though, where we want to access a property or a method of a component directly using local template reference variables.

Local reference variables in templates

We have seen how we can 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**, are used to 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 event 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 itself opens the door to its public API.

The public API of a component consists of all `public` and `protected` 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
  #product
  [name]="selectedProduct"
  (bought)="onBuy()"
></app-product-detail>
```

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

```
<span>{{product.name}}</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 used when we do not have control over the child component to add input or output binding properties.

We have mainly explained how the component class interacts with its template or other components, but we have barely been concerned about their styling.

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 `styleUrls` 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`

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

On top of that, Angular embeds style sheets at the head of a document so that they may affect other elements of our application. To prevent this from happening, we can set up different levels of view encapsulation.

View encapsulation is how Angular needs to manage CSS scoping within the component for both Shadow DOM-compliant browsers and those without support. It can be changed by setting the `encapsulation` property of the `@Component` decorator in one of the following `ViewEncapsulation` enumeration values:

- `Emulated`: This is the default option and 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.
- `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.

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

1. Open the `product-detail.component.css` file and add a CSS style to change the color of the `<h2>` element:

```
h2 {
  border: 2px dashed black;
}
```

2. Run the application using the `ng serve` command and notice that the **Product Details** caption has a black dashed border around it:

Figure 4.8: Default view encapsulation

The **Product List** section, which is also an `<h2>` element, was not affected by the style because the default encapsulation scopes all CSS styles defined to the specific component.

3. Now, open the `product-detail.component.ts` file and set the component encapsulation to `None`:

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

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

4. The browser refreshes our application, and the resulting page now looks like the following:

Figure 4.9: No view encapsulation

In the preceding image, the CSS style has leaked up to the component tree and has affected the `<h2>` element of the product list component. The **Product List** section is now also surrounded by a dashed border.

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

Another property of the `@Component` decorator that is not widely used but is very powerful is the change detection strategy.

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 on a button, an asynchronous request is completed, or a `setTimeout` and `setInterval` method is executed. Angular **monkey patches** these event types 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 `name` 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. Let's explore a scenario where we can use a change detection mechanism:

1. Open the `product-detail.component.ts` file and create a `getter` property that returns the current product name and prints a message in the browser console:

```
get productName(): string {
  console.log('Get ${this.name}');
  return this.name;
}
```

2. Open the `product-detail.component.html` file and use the interpolation syntax to display the `productName` property in a `<span>` element:

```
<span>{{productName}}</span>
```

3. Run the application using the `ng serve` command, select a product from the list, and notice the console output of your browser. You will notice that it displays the console message from the getter property *twice* per product selection. The preceding behavior is caused by the fact that change detection is also triggered twice – once when the component is initialized and again when the `name` property changes due to user selection.

4. Modify the `@Component` decorator of the product detail component by setting the `changeDetection` property to `ChangeDetectionStrategy.OnPush`:

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

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

   The preceding change will trigger the change detection mechanism only when the reference of the `name` input property changes.

5. After the browser refreshes the application, try to select some products from the list, and you will notice that the console message now appears *once* per selection.

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. The change detection strategy concludes our journey on how we can configure a component, but the Angular framework does not stop there. It also allows us to hook into specific times in the lifecycle of a component, as we'll learn in the following section.

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 of valuable help 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 comes with an interface that defines 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 it 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:

- **OnInit**: This is called when a component is initialized
- **OnDestroy**: This is called when a component is destroyed
- **OnChanges**: This is called when values of input binding properties in the component change
- **AfterViewInit**: This is called when Angular initializes the view of the current component and its child components

All of the previous 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.io/guide/lifecycle-hooks`.

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

Performing component initialization

The `OnInit` lifecycle hook implements the `ngOnInit` method, which is called during the component initialization. At this stage, all input bindings and data-bound properties have been set appropriately, 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 understand the previous concept using the following example:

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

```
constructor() {
  console.log('Name is ${this.name} in the constructor');
}
```

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

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

3. Add the `OnInit` artifact in the list of the `ProductDetailComponent` class implemented interfaces:

```
export class ProductDetailComponent implements OnInit
```

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

```
ngOnInit(): void {
  console.log('Name is ${this.name} in the ngOnInit');
}
```

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

```
selectedProduct = 'Microphone';
```

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

```
Name is in the constructor
Name is Microphone in the ngOnInit
```

Figure 4.10: Console output

The first log message from the `constructor` contains an empty string as the value of the `name` property. The preceding behavior is because when the `undefined` value is converted to a string using the text interpolation syntax, it is automatically converted to an empty string. In the second log message, the value of the `name` 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 `OnInit` 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 6*, *Managing Complex Tasks with Services*.

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

Cleaning up component resources

The interface we use to hook on the destruction event of a component is the `OnDestroy` lifecycle hook, which implements the respective `ngOnDestroy` method:

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

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

  ngOnDestroy(): void {
  }
}
```

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

- Using **structural directives**, which we will learn about in *Chapter 5*, *Enrich Applications Using Pipes and Directives*
- Navigating away from a component using the Angular router, which we will learn about in *Chapter 9*, *Navigate through Application 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 7*, *Being Reactive Using Observables and RxJS*

We have already learned to pass data down to a component using an input binding. The Angular framework provides the `OnChanges` lifecycle hook, which we can use to inspect when the value of such a binding has changed.

Detecting input binding changes

The OnChanges 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` artifacts in the `product-detail.component.ts` file:

```
import { Component, Input, Output, EventEmitter, OnChanges,
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 type `SimpleChanges` 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['name'];
  const oldValue = product.previousValue;
  const newValue = product.currentValue;
  console.log('Product changed from ${oldValue} to ${newValue}');
}
```

The previous snippet tracks the `name` 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:

```
Product changed from undefined to
Angular is running in development mode. Call enableProdMode() to enable production mode.
Product changed from to Wireless Keyboard
```

Figure 4.11: Console output

In the preceding image, look closely at the first and third lines. In the latter, we can see when the log message selected the **Wireless Keyboard** product from the list. You will also notice an additional log message in the first line, stating that the product was changed from undefined to an empty string. Why is that?

The `OnChanges` lifecycle event is triggered once the value is first set and in all subsequent changes that occur through the binding mechanism. Initially, the `oldValue` is undefined since the property has not been set yet. The `newValue` is the first value we set in the property – in our case, an empty string that comes from the initial value of the `selectedProduct` property of the product list component. To eliminate the unnecessary log, we can check whether this is the first change using the `isFirstChange` method:

```
ngOnChanges(changes: SimpleChanges): void {
  const product = changes['name'];
  if (!product.isFirstChange()) {
    const oldValue = product.previousValue;
    const newValue = product.currentValue;
    console.log('Product changed from ${oldValue} to ${newValue}');
  }
}
```

If we refresh the browser now, we can see the correct message in the console window.

The last lifecycle event of an Angular component we will explore in the following section is the `AfterViewInit` hook.

Accessing child components

The `AfterViewInit` lifecycle hook of an Angular component is called when both of the following have been completed:

- The HTML template of the component has been initialized
- The HTML templates of all child components have been initialized

We can explore how the `AfterViewInit` 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. Add a new `import` statement to import the `ProductDetailComponent` class:

```
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
```

3. Create the following `productDetail` property in the `ProductListComponent` class:

```
@ViewChild(ProductDetailComponent) productDetail:
ProductDetailComponent | undefined;
```

   In the preceding snippet, we defined the property type as `ProductDetailComponent` or `undefined`. The latter is needed because the Angular framework operates in **strict mode** by default. In strict mode, Angular makes sure that we use strong typing in our Angular applications as much as possible. Strong typing detects bugs early before deployment and helps us to avoid defects in our applications.

   We have already learned how to query a component class from an HTML template using local reference variables. Alternatively, we can use the `@ViewChild` decorator to query a child component from the parent component class. The `@ViewChild` decorator is an Angular property decorator that accepts the type of component we want to query as a parameter.

4. Modify the definition of the `ProductListComponent` class so that it implements the `AfterViewInit` interface:

```
export class ProductListComponent implements AfterViewInit
```

5. The `AfterViewInit` interface implements the `ngAfterViewInit` method, which we can use to access the `productDetail` property:

```
ngAfterViewInit(): void {
  if (this.productDetail) {
    console.log(this.productDetail.name);
  }
}
```

In the preceding method, we first check if the `productDetail` property has been set because we have already declared it as `undefined`. When we query the `productDetail` property, we get an instance of a `ProductDetailComponent` class. We can then access any member of its public API, such as the `name` property.

The `AfterViewInit` 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 the structure of Angular components and the different ways to create them. We learned how to create a standalone component or register it with an Angular module. We discussed how to isolate the 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 in a bidirectional way using property and event bindings.

We were guided through the options available in Angular for creating powerful APIs for our components, so that we can provide high levels of interoperability between components, configuring their properties by assigning either static values or managed bindings. We also saw how a component can act as a host component for another child component, instantiating the former's custom element in its template, and laying the ground for larger component trees in our applications. Output parameters give the layer of interactivity we need by turning our components into event emitters, so that they can adequately communicate with any parent component that might eventually host them in an agnostic fashion.

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 some additional insights into how we can benefit from Shadow DOM's CSS scoping on a per-component basis. 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 took a tour of the component lifecycle and learned how we can 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 extensively in your journey with Angular, directives and pipes, which we cover in the next chapter.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`

5

Enrich 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 allow us to conduct 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:

- Introducing directives
- Transforming elements using directives
- Manipulating data with pipes
- Building custom pipes
- Building custom directives

Technical requirements

The chapter contains various code samples to walk you through the concept of Angular pipes and directives. You can find the related source code in the `ch05` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Introducing 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 to it or alter its appearance. There are three types of directives:

- **Components**: They are directives with an associated template
- **Structural directives**: They add or remove elements from the DOM
- **Attribute directives**: They modify the appearance of or define a custom behavior for a DOM element

Angular provides us with a set of built-in directives that we can use in our components to cover most use cases. Angular built-in directives are part of the `CommonModule`. So, we need to import `CommonModule` when we want to use them.

The Angular CLI imports `CommonModule` by default when we create a new Angular module, as we learned in *Chapter 3*, *Organizing Application into Modules*.

We will explore the most popular directives in the following sections.

Transforming elements using directives

The Angular framework includes a set of ready-made structural directives that we can start using straight away in our applications:

- **ngIf**: Adds or removes a portion of the DOM tree based on an expression
- **ngFor**: Iterates through a list of items and binds each item to a template
- **ngSwitch**: Switches between templates within a specific set and displays each one depending on a condition

We will describe each one of them in the following sections.

Displaying data conditionally

The `ngIf` directive adds or removes an HTML element in the DOM based on the evaluation of an expression. If the expression evaluates to `true`, the element is inserted into the DOM. Otherwise, the element is removed from the DOM.

Do you recall from the previous chapter that the product details component was loaded even if there was no product selected? We can now fix that issue by binding the `*ngIf` directive to a conditional expression in the `product-detail.component.html` file:

```
<div *ngIf="name">
  <h2>Product Details</h2>
  <h3>{{name}}</h3>
  <button (click)="buy()">Buy Now</button>
</div>
```

The product detail component in the preceding HTML template is rendered on the screen when the `name` property has a value. Otherwise, it is removed completely.

Someone could reasonably point out that we could bind to the `hidden` property of the `<div>` element instead:

```
<div [hidden]="!name">
  <h2>Product Details</h2>
  <h3>{{name}}</h3>
  <button (click)="buy()">Buy Now</button>
</div>
```

The difference is that the `*ngIf` directive adds or removes elements from the DOM tree whereas the `hidden` property hides or displays existing elements in the DOM tree. The former is recommended when dealing with a large amount of data, such as lists with hundreds of items or elements that contain advanced presentation logic in their child elements. In such cases, it performs better because Angular does not need to keep data or elements in memory, runtime, as it does with the `hidden` property.

You have probably noticed the asterisk (*) character in front of `ngIf`. The asterisk indicates a structural directive, and it is syntactic sugar that acts as a shortcut for a more complicated one. Angular embeds the HTML element marked with the `*ngIf` directive into an `ng-template` element, which is used later to render the actual content on the screen. The `ng-template` is neither added to the DOM tree nor rendered on screen. Instead, it acts as a wrapper to group other HTML elements. These elements are not rendered automatically on the screen, but structural directives trigger them. Consider a scenario where we want to display a default message when there is no selected product from the list. We need to modify the `product-list.component.html` file:

```
<h2>Product List</h2>
<ul>
```

```
  <li (click)="selectedProduct = 'Webcam'">Webcam</li>
  <li (click)="selectedProduct = 'Microphone'">Microphone</li>
  <li (click)="selectedProduct = 'Wireless Keyboard'">Wireless Keyboard</
li>
</ul>
<app-product-detail
  *ngIf="selectedProduct"
  [name]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<p *ngIf="!selectedProduct">No product selected!</p>
```

If you run the application and preview it on your browser, you will see that the message is displayed when we have not selected a product from the list yet:

Figure 5.1: Product list

The approach of using multiple `*ngIf` statements has some drawbacks:

- It is error-prone because it is easy to make mistakes when composing them
- The syntax is not readable
- It goes against the **Don't Repeat Yourself (DRY)** principle

Alternatively, we can use a `<ng-template>` element to compose an `if-else` statement in the template of our component:

```
<h2>Product List</h2>
<ul>
  <li (click)="selectedProduct = 'Webcam'">Webcam</li>
  <li (click)="selectedProduct = 'Microphone'">Microphone</li>
  <li (click)="selectedProduct = 'Wireless Keyboard'">Wireless Keyboard</
li>
```

```
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [name]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

We have added another statement in the template expression of the `*ngIf` directive, which is the `else` statement of the `if-else` syntax. It is separated from the first one using a semicolon.

You can chain multiple statements in a template expression by separating them using semicolons.

The `else` statement refers to a `noProduct` variable that is activated if the condition of the `*ngIf` directive is not satisfied. The `noProduct` variable is a template reference variable, as we learned in *Chapter 4*, *Enabling User Experience with Components*, that points to an `<ng-template>` element containing a paragraph element. The paragraph element is displayed on the screen only when the `else` statement is activated.

The `ngIf` directive is useful for displaying particular pieces of the user interface. It is common to combine it with the `ngFor` directive when we want to display multiple pieces of data.

Iterating through data

The `ngFor` directive allows us to loop through a collection of items and render a template for each one, 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 ngFor as the `for` loop for HTML templates.

The product list component we already use displays a list of products using static data on the HTML template. In a real-world scenario, data is dynamic and comes from different sources, such as a backend server or the local storage of the browser.

We can use the `ngFor` directive to display the product list in a more scalable way:

1. Open the `product-list.component.ts` file and create a `products` array in the `ProductListComponent` class:

```
products = ['Webcam', 'Microphone', 'Wireless keyboard'];
```

2. Modify the unordered list element in the `product-list.component.html` file so that it displays a `<li>` element for each item from the `products` array:

```
<ul>
  <li *ngFor="let product of products" (click)="selectedProduct =
product">
    {{product}}
  </li>
</ul>
```

   In the preceding code, we use the `*ngFor` directive 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 value using Angular interpolation syntax. Finally, we bind to the `click` event of the `<li>` element only once, so the `selectedProduct` property is set to the clicked item.

The application output should be the same as before because we did not make any actual changes. We refactored the code using the `ngFor` directive to make it more manageable and scalable. However, the capabilities of the `ngFor` directive expand beyond displaying a list of data.

The `ngFor` directive 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 `*ngFor` directive using the following syntax:

```
<li *ngFor="let product of products; let variable=property"></li>
```

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

- `index`: Indicates the index of the item in the array, starting at 0 (number)
- `first`/`last`: Indicates whether the current item is the first or last item in the array (boolean)
- even/odd: Indicates whether the index of the item in the array is even or odd (Boolean)

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:

```
<ul>
  <li *ngFor="let product of products; let i=index">
    {{i+1}}. {{product}}
  </li>
</ul>
```

During the execution of `ngFor`, data may change, elements may be added or removed, and the whole list may even be replaced. Angular must take care of these changes by creating/removing elements to sync changes to the DOM tree. This is a process that can become very slow and expensive and will eventually result in the poor performance of your application.

Angular deals with variations within a collection by keeping DOM elements in memory. Internally, it uses something called **object identity** to keep track of every item in a collection. We can, however, use a specific property of the iterable items instead of the internal Angular object identity using the `trackBy` property:

```
<ul>
  <li *ngFor="let product of products; trackBy: trackByProducts">
    {{product}}
  </li>
</ul>
```

The `trackBy` property defines the `trackByProducts` method that is declared in the component class and accepts two parameters: the index of the current product and the actual product name. It returns the unique product name that we want to use as the object identity:

```
 trackByProducts(index: number, name: string): string {
  return name;
}
```

We use `ngIf` and `ngFor` most of the time during Angular development. Another structural directive that is not so commonly used is the `ngSwitch` directive.

Switching through templates

We learned that structural directives such as `ngIf` and `ngFor` are prefixed with an asterisk. The `ngSwitch` directive is an exception to this rule. It is used to switch between templates and display each one depending on a defined value.

You can think of `ngSwitch` as like an ordinary switch statement that we use in other programming languages. It consists of a set of other directives:

- `[ngSwitch]`: Defines the property that we want to check when applying the directive
- `*ngSwitchCase`: Adds or removes a template from the DOM tree depending on the value of the property defined in the `[ngSwitch]` statement
- `*ngSwitchDefault`: Adds a template to the DOM tree if the value of the property defined in the `[ngSwitch]` directive does not meet any `*ngSwitchCase` statement

We will learn how to use the directive by adding the following `<div>` element to the `product-detail.component.html` file:

```
<div *ngIf="name">
  <h2>Product Details</h2>
  <h3>{{name}}</h3>
  <div [ngSwitch]="name">
    <p *ngSwitchCase="'Webcam'">
      Product is used for video
    </p>
    <p *ngSwitchCase="'Microphone'">
      Product is used for audio
    </p>
    <p *ngSwitchDefault>Product is for general use</p>
  </div>
  <button (click)="buy()">Buy Now</button>
</div>
```

The `[ngSwitch]` directive evaluates the `name` property of the component. When it finds a match, it activates the appropriate `*ngSwitchCase` statement. If the value of the `name` property does not match any `*ngSwitchCase` statement, the `*ngSwitchDefault` statement is activated.

Directives transform HTML elements by affecting their structure, behavior, and display. On the other hand, pipes transform data and template bindings.

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 is pretty simple, consisting of the pipe name following the expression we want to transform, separated by a pipe symbol (|). Pipes are usually used with interpolation in Angular templates and can be chained to each other. Angular has a wide range of pipe types already baked into it.

Before moving on to explore some of the built-in Angular pipes, we will make a change to our Angular working application to demonstrate their use better. The `products` array in the product list component is currently an array of string values. We will convert it into an array of product objects using interfaces:

1. Run the following Angular CLI command inside the `src\app\products` folder to create a `product` interface:

```
ng generate interface product
```

You might be surprised that we define an `interface` for our model entity rather than a `class`. This is perfectly fine when the model does not feature any business logic requiring the implementation of methods or data transformation in a constructor or setter/getter function. When the latter is not required, an interface suffices since it provides the static typing we require in a simple and lightweight fashion.

2. Open the `product.ts` file that was generated from the command we ran in the previous step and modify its content as follows:

```
export interface Product {
  name: string;
  price: number;
}
```

3. Open the `product-detail.component.ts` file and add the following `import` statement:

```
import { Product } from '../product';
```

4. Rename the input property binding to `product` and also change its type:

```
@Input() product: Product | undefined;
```

5. Also, modify the `ngOnChanges` method to accommodate the new type:

```
ngOnChanges(changes: SimpleChanges): void {
```

```
    const product = changes['product'];
    if (!product.isFirstChange()) {
      const oldValue = product.previousValue.name;
      const newValue = product.currentValue.name;
      console.log('Product changed from ${oldValue} to ${newValue}');
    }
  }
```

6. Open the `product-detail.component.html` file and modify its content as follows:

```
<div *ngIf="product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <div [ngSwitch]="product.name">
    <p *ngSwitchCase="'Webcam'">
      Product is used for video
    </p>
    <p *ngSwitchCase="'Microphone'">
      Product is used for audio
    </p>
    <p *ngSwitchDefault>Product is for general use</p>
  </div>
  <button (click)="buy()">Buy Now</button>
</div>
```

Now that we have refactored the product detail component, we need to make similar changes to the product list component:

1. Open the `product-list.component.ts` file and add the following `import` statement:

```
import { Product } from '../product';
```

2. Change the type of the `selectedProduct` and `products` properties to accommodate the new interface:

```
selectedProduct: Product | undefined;
products: Product[] = [
  {
    name: 'Webcam',
    price: 100
  },
```

```
  {
    name:  'Microphone',
    price: 200
  },
  {
    name: 'Wireless keyboard',
    price: 85
  }
];
```

3. Modify the `ngAfterViewInit` and `onBuy` methods so that there are no errors due to the new `Product` type:

```
ngAfterViewInit(): void {
  if (this.productDetail) {
    console.log(this.productDetail.product);
  }
}

onBuy() {
  window.alert('You just bought ${this.selectedProduct?.name}!');
}
```

4. Finally, open the `product-list.component.html` file and modify it accordingly:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of products" (click)="selectedProduct =
product">
    {{product.name}}
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

If you run the application using the `ng serve` command, it should still work as before. We can now start learning about the various Angular built-in pipes:

- `uppercase`/`lowercase`: They transform a string into a particular case. Experiment with the following snippet in the `product-list.component.html` file to display the product name in uppercase and lowercase letters, respectively:

```
{{product.name | uppercase}}
{{product.name | lowercase}}
```

- `percent`: This formats a number as a percentage. For example, the output of `<p>{{0.1234 | percent}}</p>` will be **12%**.
- `currency`: This formats a number as a local currency. We can override local settings and change the symbol of the currency, passing the currency code as a parameter to the pipe. Open the `product-detail.component.html` file and add the following `<span>` element after the product name to display the price of the selected product in *euros*:

```
<span>{{product.price | currency:'EUR'}}</span>
```

  If you click on the **Microphone** product from the list, it should display the following data in the **Product Details** section:

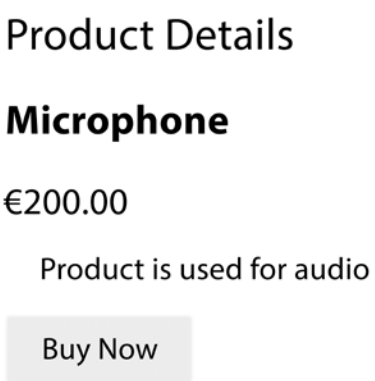


Figure 5.2: Product details

- `slice`: This subtracts a subset (slice) of a collection or string. It accepts a starting index, where it will begin slicing the input data, and optionally an end index as parameters. 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 following snippet displays the second and the third product from the `products` array of the product list component:

```
<ul>
  <li *ngFor="let product of products | slice:1:3">
    {{product.name}}
  </li>
</ul>
```

The `slice` pipe transforms immutable data. The transformed list is always a copy of the original data even when it returns all items.

- `date`: This formats a date or a string as a particular date format. The time zone of the formatted output is in the local time zone of the end user's machine. The following snippet displays the component property today as a date:

```
<p>{{today | date}}</p>
```

The `today` property is an object that has been initialized using the `Date` constructor:

```
today = new Date();
```

The default usage of this pipe displays the date according to the local settings of the user's machine. Still, we can pass additional formats that Angular has already baked in as parameters. For example, to display the date in a full date format, we write the following snippet:

```
<p>{{today | date:'fullDate'}}</p>
```

- `json`: This is probably the most straightforward in its definition; it takes an object as an input and outputs it in JSON format. Experiment with this pipe by adding the following snippet in the `product-detail.component.html` file:

```
<p>{{product | json}}</p>
```

The preceding snippet will display the properties of the `product` component property in JSON format, replacing single quotes with double quotes. So, why do we need this? The main reason is debugging; it's an excellent way to see what a complex object contains and print it nicely on the screen.

Remember to always use the `json` pipe when interpolating an object. If you fail, you will see the famous `[object Object]` on the screen when trying to use it.

- `async`: This is used when we manage data that is 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.
- `keyvalue`: This converts an object into a collection of key-value pairs where the `key` of each item represents the object property and the `value` is its actual value. The `keyvalue` pipe is really handy when we want to iterate over object properties using the `ngFor` directive. Suppose that the `products` property in the `ProductListComponent` class was the following object:

```
products = {
  'Webcam': 100,
  'Microphone': 200,
  'Wireless keyboard': 85
};
```

The `*ngFor` directive in the template of the product list component should be modified accordingly to display the name of each product:

```
<ul>
  <li *ngFor="let product of products | keyvalue">
    {{product.key}}
  </li>
</ul>
```

Built-in pipes and directives are sufficient for most use cases. In other cases, we must apply complex transformations to our data or templates. The Angular framework provides a mechanism to create unique customized pipes and directives. We'll learn how to generate custom pipes in the following sections.

Building custom pipes

We have already seen what pipes are and what their purpose is in the overall Angular ecosystem. Next, we are going to 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 name.

Sorting data using pipes

To create a new pipe, we use the `generate` command of the Angular CLI, passing the word `pipe` and its name as parameters:

```
ng generate pipe sort
```

When we run the preceding command in the `src\app\products` folder, it will create the `sort.pipe.ts` file and its corresponding unit test file, `sort.pipe.spec.ts`. It will also register the pipe with the associated `ProductsModule` in the `products.module.ts` file:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ProductListComponent } from './product-list/product-list.
component';
import { ProductDetailComponent } from './product-detail/product-detail.
component';
import { SortPipe } from './sort.pipe';

@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent,
    SortPipe
  ],
  imports: [
    CommonModule
  ],
  exports: [ProductListComponent]
})
export class ProductsModule { }
```

A pipe is added in the `declarations` array of an Angular module similar to an Angular component because it also needs a compilation context to execute. However, pipe files are not created inside a dedicated folder, but rather inside the folder where we run the `generate` command.

A pipe is a TypeScript class marked with the `@Pipe` decorator that implements the `PipeTransform` interface as 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 only required property in the `@Pipe` decorator is the `name` of the pipe. A pipe must implement the `transform` method of the `PipeTransform` interface that 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.

Angular has configured the `transform` method to use a particular type called `unknown`, which works similarly to `any`. 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.

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 `import` 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. Remove the `args` parameter since we want to sort explicitly by the product name.
4. Change the return type of the method 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[]): Product[] {
    return [];
  }

}
```

We are now ready to implement the sorting algorithm of our method. We use the native `sort` method of the array prototype that sorts items alphabetically by default. We 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[]): Product[] {
  if (value) {
    return value.sort((a: Product, b: Product) => {
      if (a.name < b.name) {
        return -1;
      } else if (b.name < a.name) {
        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 **Array.prototype.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 just need to call it from our component template to see it in action. Open the `product-list.component.html` file and add the pipe in the statement of the `*ngFor` directive:

```
<ul>
  <li *ngFor="let product of products | sort" (click)="selectedProduct =
product">
    {{product.name}}
  </li>
</ul>
```

If we run the application using the `ng serve` command, we will notice that the product list is now sorted by name alphabetically:

Figure 5.3: Product list sorted by name

We should mention that when using pipes with other properties of the `ngFor` directive, such as `index`, the pipe must be placed *after* the declaration of the array:

```
<ul>
  <li *ngFor="let product of products | sort; let i=index"
(click)="selectedProduct = product">
    {{product.name}}
  </li>
</ul>
```

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 them to `false` explicitly using the `pure` property in the `@Pipe` decorator:

```
@Pipe({
  name: 'sort',
  pure: false
})
```

Why would we do that in the first place? Well, there are situations where this might be necessary. 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 `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—pipes that are impure call the `transform` method every time the change detection cycle is triggered. So, this might be bad 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,
  {
    name: 'Headphones',
    price: 55
  }
];
```

In the preceding snippet, we create a new reference of the `products` array by appending a new item to the reference of the existing array.

We have already learned that Angular pipes need a compilation context from a respective Angular module. However, we can create pipes that can stand independently without a module.

Creating standalone pipes

We have already learned about standalone components in *Chapter 4*, *Enabling User Experience with Components*. Similarly, we can create Angular pipes that are not registered with an Angular module and are standalone. To create a standalone pipe using the Angular CLI, we pass the `standalone` option in the `generate` command as follows:

```
ng generate pipe filter --standalone
```

The preceding command will create the appropriate pipe files but will not modify any Angular module. The `filter.pipe.ts` TypeScript file of the generated pipe looks like the following:

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

@Pipe({
  name: 'filter',
  standalone: true
})
export class FilterPipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }

}
```

The `@Pipe` decorator contains an additional `standalone` property that indicates whether the pipe is standalone or not.

Remember that you should never add a standalone pipe to the `declarations` array of an Angular module because that would register it with that module.

We can consume a standalone pipe in an Angular module by adding it to the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent
  ],
```

```
  imports: [
    BrowserModule,
    FilterPipe
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Creating custom pipes allows us to transform our data in a particular way according to our needs. If we also want to transform template elements, we must create custom directives.

Building custom directives

Custom directives encompass a vast world of possibilities and use cases, and we would need an entire book to showcase all the intricacies and possibilities they offer. In a nutshell, they allow you to attach advanced behaviors to elements in the DOM or modify their appearance.

If a directive has a template attached, then it becomes a component. In other words, components are Angular directives with a view. This rule comes in handy when we want to decide whether we should create a component or a directive for our needs. If we need a template, we create a component; otherwise, we make it a directive.

As we have already learned, directives mainly fall into two categories: structural and attribute. In the following sections, we will showcase how to create a directive for each category from scratch.

Displaying dynamic data

We have all 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 an about 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 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 `generate` command of the Angular CLI to create a copyright directive. We pass the word `directive` and the name of the directive as parameters:

```
ng generate directive copyright
```

The preceding command will create the directive file, `copyright.directive.ts`, along with the accompanying unit test file, `copyright.directive.spec.ts`, and register it with the main application module, `app.module.ts`:

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

import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';
import { CopyrightDirective } from './copyright.directive';

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

A directive is added in the `declarations` array of an Angular module, similar to an Angular component. Directive files are created inside the folder where we run the `generate` command, similar to pipes. In this case, all related directive files are created inside the `src\app` folder of our Angular project.

A directive is a TypeScript class marked with the `@Directive` decorator that contains a `selector` property as defined in the `copyright.directive.ts` file:

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

@Directive({
  selector: '[appCopyright]'
})
export class CopyrightDirective {
```

```
  constructor() { }

}
```

The `selector` can be any valid CSS selector and works similarly to the component selector. The only difference is that we surround it in square brackets. Be aware, though, that we use it without them in an HTML template, such as:

```
<p appCopyright></p>
```

We use a custom prefix in attribute directives to minimize the risk of conflict with an HTML native attribute or another directive from a third-party library. As we learned in *Chapter 4*, *Enabling User Experience with Components*, the prefix can be customized using the Angular CLI when creating the Angular application.

The custom logic of our directive is summarized inside the `constructor`:

```
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.';
  }

}
```

We use 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 apply the following transformations to that element:

- We add the `copyright` class using the `add` method of the `classList` property. The class is defined in the `styles.css` file that exists in the `src\app` folder:

```
.copyright {
  background-color: lightgray;
  padding: 10px;
  font-family: Verdana, Geneva, Tahoma, sans-serif;
}
```

- We change the text of the element by modifying the `textContent` property.

The `ElementRef` is an Angular built-in 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 6*, *Managing Complex Tasks with Services*.

When creating directives, it is important to think about reusable functionality that doesn't necessarily relate to a particular feature. The topic chosen previously 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

The Angular framework provides two helpful decorators that we can use in our directives to enhance their functionality:

- `@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 preceding decorators are similar to the property and event binding we learned about in *Chapter 4*, *Enabling User Experience with Components*.

The native input HTML element can support different 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 to solve this problem:

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 to bind to the `class` property of the input element:

```
@HostBinding('class') currentClass = '';
```

4. Define an `onKeyPress` method using the `@HostListener` decorator to bind to the `keypress` native event of the input element:

```
@HostListener('keypress', ['$event']) onKeyPress(event:
KeyboardEvent) {
  const charCode = event.key.charCodeAt(0);
  if (charCode > 31 && (charCode < 48 || charCode > 57)) {
    this.currentClass = 'invalid';
    event.preventDefault();
  } else {
    this.currentClass = 'valid';
  }
}
```

When the user presses a key inside the input element, Angular knows to call the `onKeyPress` method because we have registered it with the `@HostListener` decorator. The `@HostListener` decorator accepts two parameters:

- **eventName**: The name of the triggered event
- **args**: A list of arguments to pass in the appropriate method upon triggering the event

In our case, we pass the `keypress` event name and the `$event` argument, respectively. The `$event` is the current event 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 `$event` object, convert it into a Unicode character using the `charCodeAt` method of the string prototype, and check it against non-numeric code. If the character is non-numeric, we call the `preventDefault` method of the `$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 the input element, `valid` if the key is numeric and `invalid` if it is not. Both classes are defined in the `styles.css` file of the `src\app` folder:

```
.valid {
  border-bottom: solid green;
}

.invalid {
  border-bottom: solid red;
}
```

Everything is now in place, and our directive looks and works great! We can validate its usage by adding it to an input element in an HTML template, such as:

```
<input appNumeric />
```

We can only type numeric values and the validity of each value is indicated by an appropriate color.

Let's summarize this section on creating attribute directives by learning how we can use them to load an Angular component dynamically.

Creating components dynamically

As we have learned in *Chapter 4*, *Enabling User Experience with Components*, the Angular framework knows how and where to load a component using its selector. When we run the `ng serve` command, the Angular compiler traverses the application component tree and tries to identify every component selector in HTML. It does that by matching tags that are unknown HTML elements with Angular components through the selector. The compiler throws an error if a match cannot be found.

The main reason the Angular compiler cannot match a selector with a component is that we have not registered it with a module, or it is not standalone. In all the examples we have seen, the component selectors matched with an Angular component directly. However, there are cases where we would like to load Angular components at runtime that are not known beforehand.

The Angular framework can load a component dynamically without adding its selector to an HTML template. However, we need to have an anchor so that Angular knows where to place the HTML template of the component. We can define an HTML element that will serve as the host of the dynamically created component. We will also attach an attribute directive to the anchor HTML element that will do most of the heavy lifting to create and add the component to the HTML template. We will explore how to work with dynamically created components by loading the details of a product that is not on our product list:

1. Run the following Angular CLI command inside the `src\app\products` folder:

   ```
   ng generate directive productHost
   ```

   The preceding command creates the `product-host.directive.ts` and `product-host.directive.spec.ts` files, although we passed the `productHost` as the directive name. When we pass camel case names in the `generate` command, the Angular CLI converts the names of the generated files to kebab case.

2. Open the `product-list.component.html` file and add the following HTML snippet:

   ```
   <h3>Offers</h3>
   <ng-template appProductHost></ng-template>
   ```

   In the preceding snippet, the `<ng-template>` component will host the dynamically created product detail component.

3. Open the `product-host.directive.ts` file and modify the `import` statements as follows:

   ```
   import { Directive, OnInit, ViewContainerRef } from '@angular/core';
   import { ProductDetailComponent } from './product-detail/product-
   detail.component';
   ```

   The `ViewContainerRef` artifact will give us access to the view container that will host the dynamically created component.

4. Modify the `ProductHostDirective` class by implementing the `ngOnInit` method of the `OnInit` interface:

   ```
   export class ProductHostDirective implements OnInit {

     constructor(private vc: ViewContainerRef) { }
   ```

```
  ngOnInit(): void {
    const productRef = this.
vc.createComponent(ProductDetailComponent);
    productRef.setInput('product', {
      name: 'Optical mouse',
      price: 130
    });
  }

}
```

In the preceding code, we inject the `ViewContainerRef` Angular service into the `constructor` to make it available to our directive. Then, we call its `createComponent` method, passing the type of the `ProductDetailComponent` class as a parameter to create a product detail component. Finally, we use the `setInput` method in the created component reference to pass a value to the input property binding of the product detail component.

5. Run the application using the `ng serve` command, and you should see the dynamically created component in the **Offers** section:

Figure 5.4: Dynamically created component

Creating components during runtime is important when we do not want to load a component upfront or when we do not know the location in which it will be loaded beforehand.

We have already seen various use cases for attribute directives. In the following section, we will get our hands dirty with structural directives that are not so widely used.

Toggling templates dynamically

A typical scenario in enterprise Angular applications is that users should have access to certain application parts according to their role. You may think that we could use the `ngIf` built-in directive for this. It would be valid for a simple case, but usually, checking a role involves calling some service to get the current user and extracting their role. We will learn about services in *Chapter 6*, *Managing Complex Tasks with Services*. For now, we could create a more specific structural directive:

1. Run the following Angular CLI command to create a permissions directive:

```
ng generate directive permission
```

2. Import the `Input`, `TemplateRef`, `ViewContainerRef`, and `OnInit` artifacts from the `@angular/core` npm package:

```
import { Directive, Input, TemplateRef, ViewContainerRef, OnInit }
from '@angular/core';
```

3. Similarly to components, we can use an `@Input` decorator in a directive if we want to pass data to our directive. Open the `permission.directive.ts` file and use the `@Input` decorator to pass the list of available roles that are eligible to access the host element. The role of the current user is hardcoded inside the directive for the sake of simplicity:

```
@Input() appPermission: string[] = [];
private currentRole = 'agent';
```

4. We can use the directive in any HTML template as follows:

```
<div *appPermission="['admin', 'agent']"></div>
```

   The name of the input property must have the same name as the selector of the directive. Notice the use of the asterisk in front of the directive. If you omit it, the Angular framework throws an error.

   If we want to add another input property, we should name it differently. The `@Input` decorator accepts an optional parameter that is the name with which the property is exposed to the public API:

```
@Input('anotherProperty') propertyName;
```

   The directive should use the `propertyName` variable for internal purposes, whereas components that use the directive should use `anotherProperty`.

5. We now need to add the business logic that adds or removes the embedded view of the host element in the DOM according to the roles that we pass in the input property:

```
export class PermissionDirective implements OnInit {

  @Input() appPermission: string[] = [];
  private currentRole = 'agent';

  constructor(private tmplRef: TemplateRef<any>, private vc:
ViewContainerRef) { }

  ngOnInit(): void {
    if (this.appPermission.indexOf(this.currentRole) === -1) {
      this.vc.clear();
    } else {
      this.vc.createEmbeddedView(this.tmplRef);
    }
  }

}
```

In the preceding code, we inject the `TemplateRef` and `ViewContainerRef` services to help us. The `TemplateRef` represents the Angular-generated `ng-template` element of the embedded view. The `ViewContainerRef` is the container used to insert the embedded view, which is adjacent to the host element.

The `ngOnInit` method first checks whether the `currentRole` belongs to the list of roles we pass as an input parameter. If it does not, it calls the `clear` method of `ViewContainerRef` to remove the host element from the DOM. Otherwise, it calls the `createEmbeddedView` method to create an embedded view of the host element inside the view container and adds it to the DOM.

In a real-world scenario, we would not hardcode the current role into the directive but use an Angular service to fetch it. The service would probably access the local storage of the browser or call an API method to a backend.

You can now test the directive by adding it to an HTML template, toggling the current role, and watching how the directive performs when adding/removing elements from the DOM.

We have already learned that Angular pipes can be standalone without needing a respective module. Similarly, we can create standalone directives without registering them with an Angular module.

Creating standalone directives

To create a standalone Angular directive using the Angular CLI, we use the `generate` command for creating directives, and we pass the `standalone` option after the directive name as a parameter:

```
ng generate directive autofocus --standalone
```

The preceding command will create the `autofocus` directive file and its accompanying unit test but will not interact with an Angular module. The `autofocus.directive.ts` TypeScript file of the directive looks like the following:

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

@Directive({
  selector: '[appAutofocus]',
  standalone: true
})
export class AutofocusDirective {

  constructor() { }

}
```

The `@Directive` decorator sets the `standalone` property to indicate that the directive is a standalone one.

Remember that you should never add a standalone directive to the `declarations` array of an Angular module because that would register it with that module.

We can consume a standalone directive in an Angular module by adding it to the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent
```

```
  ],
  imports: [
    BrowserModule,
    AutofocusDirective
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

When we combine standalone directives and pipes with standalone components, we get a clear and flexible way to structure and build our Angular applications.

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 components tree, consume data services, and leverage the new Angular router to show and hide components when required.

Nevertheless, this chapter is the backbone of Angular development, and we hope you enjoyed it as much as we did when writing about pipes and directives. 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.

6

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.

In this chapter, we'll investigate the advantages that Angular's dependency management mechanism can bring to the game 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 learn the following concepts about Angular services:

- Introducing the 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 `ch06` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-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 the **injector**. The 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 5*, *Enrich Applications with Pipes and Directives*, we learned how to display a list of objects using the `ngFor` directive. We used a static list of `Product` objects that were declared in the `ProductListComponent` class, as shown here:

```
products: Product[] = [
  {
    name: 'Webcam',
    price: 100
  },
  {
    name:  'Microphone',
    price: 200
  },
  {
    name: 'Wireless keyboard',
    price: 85
  }
];
```

The 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 service.
- The list of products 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. We will create an Angular service that will return the list of products by itself. 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 `generate` command of the Angular CLI while passing the name of the service as a parameter:

```
ng generate service products
```

Running the preceding command in the `src\app\products` folder will create the `products` service that consists of the `products.service.ts` file and its accompanying unit test file, `products.service.spec.ts`. However, the command will not modify the `products` module as someone would expect. Why is that?

Angular services are available application-wide by default and are not tied to any Angular module. We created the service inside the `products` folder to keep it together logically with the rest of the products feature. It will contain the business logic for the products feature, and it will be easier to change it if refactoring is necessary.

We usually name a service after the functionality that it represents. Every service has a business context or domain that 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, customers should be managed by a separate customers 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`.

An Angular service, by default, is not registered with a specific module like components, directives, and pipes. Instead, it 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 `getProducts` method in the `ProductsService` class:

   ```
   getProducts(): Product[] {
     return [
       {
         name: 'Webcam',
         price: 100
       },
       {
         name:  'Microphone',
         price: 200
       },
       {
         name: 'Wireless keyboard',
         price: 85
       }
     ];
   }
   ```

3. Open the `product-list.component.ts` file and modify the `products` property so that it is initialized to an empty array:

```
products: Product[] = [];
```

4. Import the `ProductsService` class that we created in step 1:

```
import { AfterViewInit, Component, ViewChild } from '@angular/core';
import { ProductDetailComponent } from '../product-detail/product-
detail.component';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

5. Create a component property called `productService` and give it a type of `ProductsService`:

```
selectedProduct: Product | undefined;
@ViewChild(ProductDetailComponent) productDetail:
ProductDetailComponent | undefined;
products: Product[] = [];
private productService: ProductsService;
```

6. Instantiate the property using the `new` keyword in the component's `constructor`:

```
constructor() {
  this.productService = new ProductsService();
}
```

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

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

8. Add the `OnInit` interface to the list of implemented interfaces of the `ProductListComponent` class:

```
export class ProductListComponent implements OnInit, AfterViewInit {
```

9. Call the `getProducts` method of `productService` inside the `ngOnInit` method and assign the return value to the `products` component 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:

Figure 6.1: Product list

Awesome! We have successfully wired up our component with the service, and our application looks great. Well, it seems this is the case, but it's not. There are some problems with the actual implementation. If the `ProductsService` class must change, maybe to accommodate for 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.

Dependency injection 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` as follows:

```
constructor(private 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 or mock it up if we wish.

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 an actual injector that can introspect the tokens used to annotate the parameters in the `constructor` of an Angular artifact.

It returns a singleton instance of the type represented by each dependency so that we can use it straight away in the implementation of our 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 startup, 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 when created with the CLI, by default. The root injector creates singleton services that are globally available through the application.

In *Chapter 3*, *Organizing Application into Modules*, we learned that the `@NgModule` decorator has a `providers` property where we can register services. Registering a service in this way is the same as configuring the service with `providedIn: 'root'` when the Angular module is imported directly from the main application module. 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 Angular services that are not used and delete them, resulting in a smaller bundle.

When you provide a service using the `@NgModule` decorator, the Angular compiler cannot say if the service is used somewhere in this module. So, it includes the service in the application bundle *a priori*. Thus, it is preferable to use the `@Injectable` decorator over the `@NgModule` one. You should always register services with the root injector unless you want to satisfy a particular case.

The root injector is not the only injector in an Angular application. Lazy-loaded modules and components have their own injectors too. The injectors of an Angular application are 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. Should the provider lookup finish with no match, it returns to the injector of the component that requested the provider and bubbles up the module injector hierarchy until it reaches the root injector. If no match is found, Angular throws an exception.

The following diagram shows how the Angular DI mechanism works:

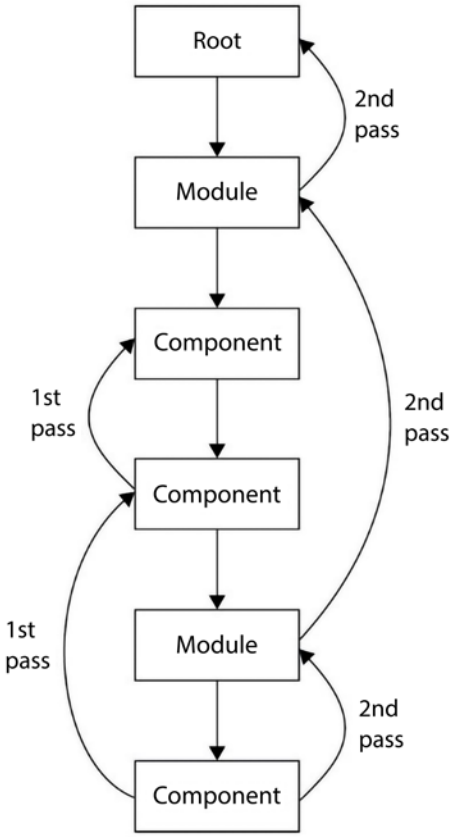


Figure 6.2: The injector tree

When a component asks for a dependency, the application enters a process that is divided into two phases, known as passes:

- **1st pass**: It searches through the injectors of all the parent components up through the component tree. If it finds the dependency, it stops and returns an instance to the component that requested it. Otherwise, it proceeds to the 2nd pass.

- **2nd pass**: It searches through the injectors of all the parent modules, including the root injector of the application. If the dependency is not found, an error is thrown. Otherwise, it returns an instance of the dependency on the component.

Components create their 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

The `@Component` decorator has a `providers` property similar to the `@NgModule` decorator to register services with a component injector. A service that registers with the component 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 the component injector can be shared among the child components of the parent component injector, and it is immediately available for injection at their constructors. Child components reuse the same instance of the service from the parent component. Let's walk our way through an example to understand this better:

1. Create a new component named `favorites` inside the `src\app\products` folder:

```
ng generate component favorites
```

2. Add the `favorites` component in the `product-list.component.html` file:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of products | sort"
(click)="selectedProduct = product">
    {{product.name}}
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
```

```
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
<h2>Favorites</h2>
<app-favorites></app-favorites>
```

3. Open the `product-list.component.ts` file and add the `ProductsService` class to the `providers` property of the `@Component` decorator:

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

4. Modify the `favorites.component.ts` file to get the list of products from `ProductsService`:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';

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

  products: Product[] = [];

  constructor(private productService: ProductsService) { }

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }

}
```

5. Finally, open the `favorites.component.html` file and replace its content with the following HTML template:

```
<ul>
  <li *ngFor="let product of products | slice:1:3">
    {{product.name}}
  </li>
</ul>
```

In the preceding template, we use the `*ngFor` directive to display the product list. Our favorite products will be a subset of the initial product list. So, we apply the `slice` pipe to display only the last two products.

When running the application using `ng serve`, you should see the following output:

Figure 6.3: Favorite product list

Let's explain what we did in the previous example in more detail. We injected `ProductsService` into the `constructor` of `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 didn'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 its parent component, `ProductListComponent`.

So, even if `ProductsService` was initially registered with the root injector, we could also register it with the 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 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 through its constructor, nothing more. Now, if the component provides the same service through its injector, it will get an instance of the service that is entirely different from the one from the root injector. The previous technique is called **service scope limiting** because we limit the scope of the service to a specific part of the component tree:

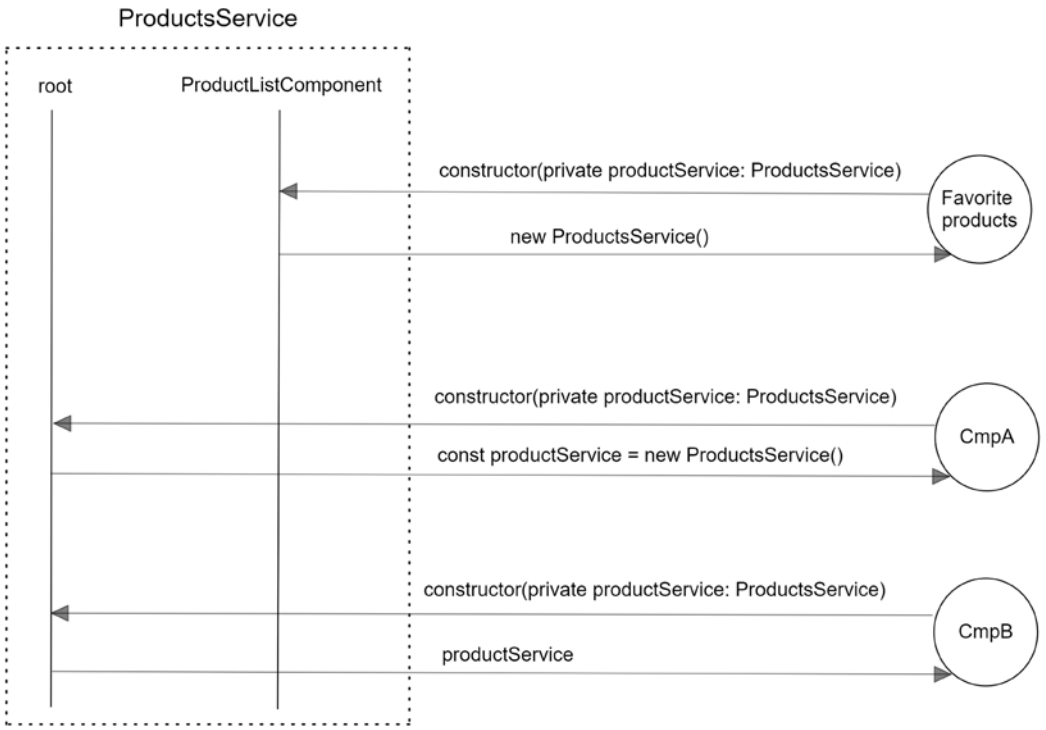


Figure 6.4: Service scope limiting

The previous diagram shows that `ProductsService` can be provided through two injectors: the application root injector and `ProductListComponent`. The `FavoritesComponent` class injects `ProductsService` into its `constructor` to use it. As we have already seen, `FavoritesComponent` is a child component of `ProductListComponent`.

According to the application injector tree we saw in the *Providing dependencies across the application* section, 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, called `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. Luckily, `ProductsService` is also registered with the root injector. The root injector 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 called `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 component 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 inside the `src\app\products` folder 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',
  templateUrl: './product-view.component.html',
  styleUrls: ['./product-view.component.css']
})
```

```
export class ProductViewComponent {
  @Input() id = -1;
}
```

3. Run the following Angular CLI command inside the `src\app\products\product-view` folder to create an Angular service that will be dedicated to the product view component:

```
ng generate service product-view
```

The location where we create a service is not related to the injector that provides it. It is just a visual representation so that we can quickly identify where it is used. If we had created the service in the folder of the `products` module, we might have deduced that it is available to the whole module.

4. Open the `product-view.service.ts` file and remove the parameter object from the `@Injectable` decorator because we will provide it later in the product view component.
5. 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.

6. Create a method named `getProduct` that takes an `id` property as a parameter. The method will call the `getProducts` method of `ProductsService` and will search through the product list based on the index of the array. 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[id];
    }
    return this.product;
  }

}
```

We set the `product` property only if it has not been set already. We will see later why it is important to do so.

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 from the `OnInit` interface:

```
import { Component, Input, OnInit } from '@angular/core';
import { ProductViewService } from './product-view.service';

@Component({
  selector: 'app-product-view',
  templateUrl: './product-view.component.html',
  styleUrls: ['./product-view.component.css'],
  providers: [ProductViewService]
})
export class ProductViewComponent implements OnInit {

  @Input() id = -1;
```

```
  constructor(private productviewService: ProductViewService) { }

  ngOnInit(): void {
  }

}
```

2. Create a `name` component property to keep the product name that we will fetch from `ProductViewService`:

```
name = '';
```

3. Modify the `ngOnInit` method so that it calls the `getProduct` method of `ProductViewService` as follows:

```
ngOnInit(): void {
  const product = this.productviewService.getProduct(this.id);
  if (product) {
    this.name = product.name;
  }
}
```

In the preceding snippet, we pass the `id` component property to the `getProduct` method as a parameter and assign the returned value to the `name` property.

4. We now only need to display the `name` property in the component template. Open the `product-view.component.html` file and replace its content with the following HTML template:

```
{{name}}
```

5. Finally, open the `product-list.component.html` file and modify the unordered list element to use the product view component:

```
<ul>
  <li *ngFor="let product of products | sort; let i=index"
(click)="selectedProduct = product">
    <app-product-view [id]="i"></app-product-view>
  </li>
</ul>
```

In the preceding snippet, we use the `index` property of the `ngFor` directive to access the index of the current product in the list. We then pass the index to the `app-product-view` component as an `id` so that it can fetch and display the related product name.

If we run our application with the `ng serve` command, we will see the following output:

Figure 6.5 – Product list

You may notice that the product list is not sorted, although we use the `sort` pipe. The preceding behavior is caused by the fact that the `index` property is applied to the `products` array directly and not to its sorted version. To overcome this behavior, we should sort the array directly in the TypeScript class of the component before using it in the `ngFor` directive.

Each product view component rendered creates a dedicated sandboxed `ProductViewService` instance for its purpose. The instance cannot be shared by any other component instance and cannot 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. 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[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.

With that, we 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 DI down the component tree

In the previous sections, we saw how `ProductListComponent` registered `ProductsService`, making it immediately available to all the child components. A component may contain child components at different levels. That is, its child components can have other child components, and so on. Sometimes, we might need to restrict dependency access to components located next to a specific component in the hierarchy. We can do that by registering the service in the `viewProviders` property of the `@Component` decorator:

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

In the preceding snippet, we define that `ProductsService` should only be accessible by the injectors of the components located in the view of `ProductListComponent`, and its children.

Restricting provider lookup

Just like restricting DI, 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 { Component, Host, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';

@Component({
  selector: 'app-favorites',
  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 injector of `FavoritesComponent` will look for the `ProductsService` class at the component providers. If it does not provide the service, it will not bubble up the injector hierarchy; instead, it will stop and throw an exception. We can configure the injector so that it does not throw an error if we decorate the service with the `@Optional` decorator:

```
import { Component, Host, OnInit, Optional } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';

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

  products: Product[] = [];

  constructor(@Host() @Optional() private productService: ProductsService)
{ }

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }

}
```

The `@Host` and `@Optional` decorators define at what level the injector searches for dependencies. There are two other decorators additional to them, 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.

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 have already learned how to use the `providers` property in an Angular decorator:

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

The preceding syntax is called **class provider** syntax and is shorthand for the **provide object literal** syntax shown below:

```
@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css'],
  providers: [
    { provide: ProductsService, useClass: ProductsService }
  ]
})
```

The provide object literal syntax consists of two properties:

- `provide`: This is the token that's 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 the `useClass` name because we are providing a class.

Let's have a 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 `favorites` service that would extend the `ProductsService` class and filter out data using the native `slice` method of the array instead of the pipe. The `favorites.service.ts` file would look like the following:

```
import { Injectable } from '@angular/core';
import { ProductsService } from './products.service';
import { Product } from './product';

@Injectable({
  providedIn: 'root'
})
export class FavoritesService extends ProductsService {

  constructor() {
    super();
  }

  override getProducts(): Product[] {
    return super.getProducts().slice(1, 3);
  }

}
```

Some things that should be pointed out in the preceding service are:

- We use the `extends` keyword to indicate that `ProductsService` is the base class of `FavoritesService`.

- The `constructor` calls the `super` method to execute any business logic inside the base class constructor.
- The `getProducts` method is marked with the `override` keyword to indicate that the implementation of the method replaces the corresponding method of the base class.

We could then provide the new service in the decorator of `FavoritesComponent` using the `useClass` syntax. The `favorites.component.ts` file should look like the following:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
import { FavoritesService } from '../favorites.service';

@Component({
  selector: 'app-favorites',
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css'],
  providers: [
    { provide: ProductsService, useClass: FavoritesService }
  ]
})
export class FavoritesComponent implements OnInit {

  products: Product[] = [];

  constructor(private productService: ProductsService) { }

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }

}
```

Notice that we did not need to change anything except for providing the new service. The remaining TypeScript code is the same as before. To see it in action, remember to remove the `slice` pipe from the `favorites.component.html` file before running the application.

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 we need, as we will learn in the following section.

Providing services conditionally

In the example of 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();
  };
}
```

We could then modify the `providers` property in the decorator of `FavoritesComponent` so that the `favorites.component.ts` file would look like this:

```
import { Component, OnInit } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
import { favoritesFactory } from '../favorites';

@Component({
  selector: 'app-favorites',
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css'],
  providers: [
    { provide: ProductsService, useFactory: favoritesFactory(true) }
  ]
})
export class FavoritesComponent implements OnInit {

  products: Product[] = [];
```

```
  constructor(private productService: ProductsService) { }

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }

}
```

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 should add it to the `deps` property of the provide object literal syntax:

```
@Component({
  selector: 'app-favorites',
  templateUrl: './favorites.component.html',
  styleUrls: ['./favorites.component.css'],
  providers: [
    {
      provide: ProductsService,
      useFactory: favoritesFactory(true),
      deps: [ProductViewService]
    }
  ]
})
```

We could then use it in the factory function of the `favorites.ts` file as follows:

```
import { FavoritesService } from './favorites.service';
import { ProductsService } from './products.service';
import { ProductViewService } from './product-view/product-view.service';

export function favoritesFactory(isFavorite: boolean) {
  return (productViewService: ProductViewService) => {
    if (isFavorite) {
      return new FavoritesService(productViewService);
    }

    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? Suppose that our application settings are defined in an `app.config.ts` file inside the `src\app` folder:

```
export interface AppConfig {
  title: string;
  version: number;
}

export const appSettings: AppConfig = {
  title: 'My application',
  version: 1.0
};
```

You may think we could provide these settings as `{ provide: AppConfig, useValue: appSettings }`, but this will throw an error because `AppConfig` 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:

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

export interface AppConfig {
  title: string;
  version: number;
}

export const appSettings: AppConfig = {
  title: 'My application',
  version: 1.0
};

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

We could then provide it in the main application component using the `providers` property of the `@Component` decorator and start using it in the `AppComponent` class:

```
import { Component, Inject } from '@angular/core';
import { APP_CONFIG, appSettings, AppConfig } from './app.config';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    { provide: APP_CONFIG, useValue: appSettings }
  ]
})
export class AppComponent {
  title = 'my-app';

  constructor(@Inject(APP_CONFIG) config: AppConfig) {}
}
```

To inject an `InjectionToken` object in an Angular component or other artifacts, we need to use the `@Inject` decorator. It accepts an `InjectionToken` object as a parameter and declares an object that matches the interface specified as generic when the token was created. In the preceding snippet, the `@Inject` decorator takes the `APP_CONFIG` token as a parameter and creates an object of `AppConfig` type.

The `useValue` syntax is particularly useful when testing Angular applications. We will use it extensively when we learn about unit testing in *Chapter 12, Unit Test an Angular Application*.

Note that although the `AppConfig` interface did not have a significant role in the injection process, we need it to provide typing on the configuration object.

The 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, which are heavily based on Angular DI.

In this chapter, we learned what Angular DI is and how we can start leveraging 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.

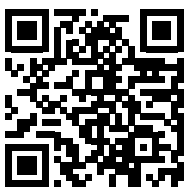
Finally, we investigated how to override Angular services by replacing the service implementation or transforming existing objects into services.

In the next chapter, we will learn what reactive programming is and how we can use observables in the context of an Angular application.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`



7

Being Reactive Using Observables and RxJS

Handling asynchronous information is a common task in our everyday lives as developers. **Reactive programming** is a programming 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 it comes to 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.

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 the concept of observables and RxJS. You can find the related source code in the `ch07` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Strategies for handling asynchronous information

We manage data asynchronously in different forms, such as consuming data from a backend API or reading contents from the local filesystem. Consuming information from an API is a typical operation in our daily development workflow. We consume data over HTTP all the time, such as when authenticating users by sending out credentials to an authentication service. We also use HTTP when fetching the latest tweets in our favorite Twitter widget. 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 the following sections, we put in place mechanisms to handle states in 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 some time has elapsed. 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 called when the asynchronous operation has been completed. Let's see how to use a callback through an example:

1. First, run the following Angular CLI command to create a new Angular application:

   ```
   ng new my-app --defaults
   ```

   In the preceding command, the `--defaults` option instructs the Angular CLI to create the Angular project with default values for routing and styling.

2. Open the `app.component.ts` file and create a `setTitle` property to change the `title` property of the component. Notice that it returns an arrow function because we are going to use it as a callback to another method:

   ```
   private setTitle = () => {
     this.title = 'Learning Angular';
   }
   ```

3. Next, create a `changeTitle` method that calls another method, named, by convention, `callback`, after 2 seconds:

```
private changeTitle(callback: Function) {
  setTimeout(() => {
    callback();
  }, 2000);
}
```

4. Finally, add a `constructor` in the component and 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 2 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 ends up looking horrible and complicated, which leads 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 2 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 do 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 why do we need another paradigm? 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're 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 notification unless we rerun everything from scratch. Moreover, we sometimes need a more proactive implementation of asynchronous data handling, which is where observables come into the picture. 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.

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 machinery it needs to produce and emit such events. It can be fired and canceled at any time, regardless of whether it has emitted the expected data already.

Observers need to 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. Replace `setTimeout` with `setInterval` in the `onComplete` method that we covered previously:

```
private onComplete() {
  return new Promise<void>(resolve => {
    setInterval(() => {
      resolve();
    }, 2000);
```

```
  });
}
```

The promise will now resolve repeatedly every 2 seconds.

2. Modify the `setTitle` property to append the current timestamp in the `title` property of the component:

```
private setTitle = () => {
  const timestamp = new Date().getMilliseconds();
  this.title = 'Learning Angular (${timestamp})';
}
```

3. Run the Angular application, and you will notice that the timestamp is set only once after 2 seconds and never changes again. The promise resolves itself, and the entire asynchronous event terminates at that very moment. It is not the desired behavior for our application so let's fix it using observables!
4. Import the `Observable` artifact from the `rxjs` npm package:

```
import { Observable } from 'rxjs';
```

5. 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 2 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.

6. Modify the `constructor` of the 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 now changes every 2 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 what we know as 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

Sound daunting? It isn't. The big challenge here is to change our mindset and learn 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 it 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 that should look like the following diagram:

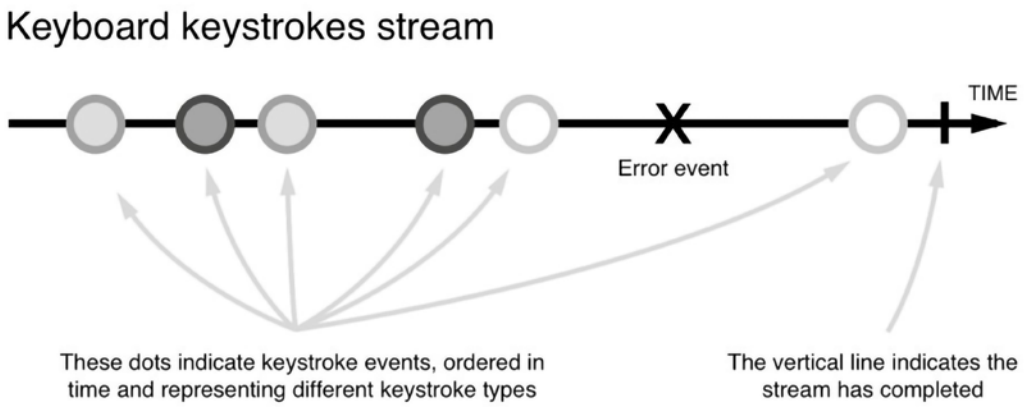


Figure 7.1: Timeline of keystroke events

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 2 seconds, another value was emitted. What's the difference between that code and our keyUp events? Nothing. We know how often a timer interval is triggered. In the case of keyUp events, we don't know because it is not under our control. But that is the only difference, which means keyUp events can also be considered observables. Let's try to explain it further by implementing a key logger in our app:

1. Create a new Angular component with the name `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` component 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:

```
@ViewChild('keyContainer', { static: true }) input: ElementRef |
undefined;
keys = '';
```

The `input` property is used to query the `<input>` HTML element using the `keyContainer` template reference variable. The second parameter of the `@ViewChild` decorator is an object with a `static` property. The `static` property indicates whether the element we want to query will be available during component initialization. In our case, the `<input>` element is already present on the DOM, so we set its value to `true`. However, there are cases where an HTML element is not present initially, such as when using the `NgIf` directive to add it conditionally. In that case, instead of setting its value to `false`, we can remove the second parameter of the `@ViewChild` decorator completely.

5. Add the following `import` statement to import the `fromEvent` artifact from the `rxjs` npm package:

```
import { fromEvent } from 'rxjs';
```

The RxJS library has a variety of helpful artifacts, called **operators**, that we can use with observables. One of them is the `fromEvent` operator, which 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 keep pressed keys in the `keys` property:

```
export class KeyLoggerComponent implements OnInit {
```

```
  @ViewChild('keyContainer', { static: true }) input: ElementRef |
undefined;
  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 using the `@ViewChild` decorator is an `ElementRef` object, which is a wrapper over the actual HTML element.

7. Open the `app.component.html` file and replace its content with the following HTML template:

```
<span>{{title}} app is running!</span>
<div>
  <app-key-logger></app-key-logger>
</div>
```

Run the application using the `ng serve` command and start pressing keys to verify the use of the key logger that we have just created:

Figure 7.2: 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 that allows us to create observables out of a large variety of scenarios, including the following:

- Interaction events
- Promises
- Callback functions
- Events

In this sense, reactive programming does not aim to replace asynchronous patterns, such as promises or callbacks. All the way around, it can leverage them as well to create observable sequences.

RxJS has built-in support for a wide range of 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 sections.

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:

```
import { of } from 'rxjs';

const values = of(1, 2, 3);
values.subscribe(value => console.log(value));
```

The previous snippet will print the numbers `1`, `2`, and `3` in the console window *in sequence*.

The `from` operator is used to convert an array or a promise to an observable:

```
import { from } from 'rxjs';

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 start migrating from promises to observables in your Angular application if you have not done so already!

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 5*, *Enrich 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 is used to link and combine 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, map, tap } 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 to exclude non-numeric values. Import the `filter` operator from the `rxjs` npm package:

```
import { filter, fromEvent, map, tap } from 'rxjs';
```

6. Add the following snippet after the `map` operator in the `ngOnInit` method:

```
filter(code => (code > 31 && (code < 48 || code > 57)) === false),
```

   In the preceding snippet, we omit the `return` statement from the arrow function. It is a shorthand syntax that requires writing the arrow function in one line without brackets.

7. The observable currently emits Unicode character codes. We need to convert them back to actual 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:

1. Add the `Input` artifact in the `import` statement of the `@angular/core` npm package:

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

2. Add a `numeric` input property in the `KeyLoggerComponent` class:

```
@Input() numeric = false;
```

3. 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();
}
```

We have already seen RxJS operators manipulating observables that return primitive data types such as numbers, strings, and arrays. However, there are additional operators that we can use to work with observables that also return observables as values.

Higher-order observables

Observables that operate on other observables *of* observables are called **higher-order observables**. Higher-order observables have an *inner* observable that contains the actual values we are interested in using. We can use specific RxJS operators to *flatten* the inner observable and extract its values. The most used flattened operators in Angular development are the `switchMap` and `mergeMap` operators.

The `switchMap` operator takes an observable as a source and applies a given function to each item, returning an inner observable for each one. The operator returns an output observable with values emitted from each inner observable. As soon as an inner observable emits a new value, the output observable stops receiving values from the other inner observables.

We will now investigate how `switchMap` is usually used in Angular applications. Remember the `ProductsService` and `ProductViewService` classes in the previous chapter? We will now convert them to use observables instead of plain arrays and learn how to combine them using the `switchMap` operator.

If you want to follow along with the source code of this chapter, make sure that you copy the `products` folder from the source code of *Chapter 6*, *Managing Complex Tasks with Services*, in the `src\app` folder of your current project. Otherwise, you can continue from where you left off with your project in the previous chapter.

In this chapter, we will work with the product list and product view components for simplicity and repeatability. However, the `products` module also contains other components and services that must be modified to reflect the use of observables in the codebase. We encourage you to make these changes yourselves. The source code described in the *Technical requirements* section has been trimmed down to compile successfully without those components and services.

Let's get started:

1. Open the `products.service.ts` file and import the `of` and `Observable` artifacts from the `rxjs` npm package:

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Product } from './product';
```

2. Extract the `products` array into a separate service property to enhance code readability:

```
private products = [
  {
    name: 'Webcam',
    price: 100
  },
  {
    name:  'Microphone',
```

```
    price: 200
  },
  {
    name: 'Wireless keyboard',
    price: 85
  }
];
```

3. Modify the `getProducts` method so that it returns the `products` property:

```
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.

4. Open the `product-view.service.ts` file and add the following `import` statement:

```
import { Observable, of, switchMap } from 'rxjs';
```

5. Modify the `getProduct` method to return an observable of a `Product` object:

```
getProduct(id: number): Observable<Product> {
  return this.productService.getProducts().pipe(
    switchMap(products => {
      if (!this.product) {
        this.product = products[id];
      }
      return of(this.product);
    })
  );
}
```

In the preceding method, there are a lot of RxJS mechanics involved. We call the `getProducts` method of the `ProductsService` class, which returns an observable of products. We also use the `pipe` operator to chain the observable of products with the observable returned from the `switchMap` operator. The `switchMap` operator creates a new inner observable for each product emitted from the source observable, using the `of` operator. Finally, the `getProduct` method returns the output observable that results from the `pipe` operator.

As the name implies, the `switchMap` operator cancels any current inner observable that is active and *switches* to a new one when the source observable emits a new value. If we would like to wait for all inner observables to complete, we could use the `mergeMap` operator from RxJS. The `mergeMap` operator, as the name implies, *merges* values from all inner observables into one. The only change we must make to start using the `mergeMap` operator is to modify the `product-view.service.ts` accordingly:

```
import { Injectable } from '@angular/core';
import { Observable, of, mergeMap } from 'rxjs';
import { ProductsService } from '../products.service';
import { Product } from '../product';

@Injectable()
export class ProductViewService {

  private product: Product | undefined;

  constructor(private productService: ProductsService) { }

  getProduct(id: number): Observable<Product> {
    return this.productService.getProducts().pipe(
      mergeMap(products => {
        if (!this.product) {
          this.product = products[id];
        }
        return of(this.product);
      })
    );
  }

}
```

We have started using observables in our Angular services using the RxJS library. However, our application is now broken because our Angular components did not reflect that change. We need to adjust them so they can interact with the new observable-based services and get data using observable streams.

Subscribing to observables

We have already learned that an observer needs to subscribe to an observable in order to start getting emitted data. Our products and product-view services currently emit product data using observables. We must modify their respective components to subscribe and get these 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 `products` 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, but we did not. Component life cycle event methods should be as clear and concise as possible. Always try to extract their logic in a separate method for clarity.

3. Open the `product-view.component.ts` file and create a `getProduct` method in the `ProductViewComponent` class:

```
private getProduct() {
  this.productviewService.getProduct(this.id).subscribe(product => {
    if (product) {
      this.name = product.name;
    }
  });
}
```

Similarly to step 1, we subscribe to the `getProduct` method of the `ProductViewService` class and set the `name` component property inside the `subscribe` method.

4. We also need to modify the `ngOnInit` method so that it calls the `getProduct` method:

```
ngOnInit(): void {
  this.getProduct();
}
```

If you are working with the Angular project you created in the previous chapter, skip steps 5 and 6.

5. Open the `app.module.ts` file and import `ProductsModule`:

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

import { AppComponent } from './app.component';
import { KeyLoggerComponent } from './key-logger/key-logger.
component';
import { ProductsModule } from './products/products.module';

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

6. Add the product list component in the `app.component.html` file:

```
<span>{{title}} app is running!</span>
```

```
<div>
  <app-key-logger></app-key-logger>
</div>
<app-product-list></app-product-list>
```

Run the application using the `ng serve` command, and you should see the product list displayed on the page successfully:

Figure 7.3: Product list

As depicted in the previous image, we have achieved the same result of displaying the product list as in *Chapter 6*, *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 that 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 has created the subscription needs to be destroyed.

Some of the most well-known techniques to use when we are concerned with 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 sections.

Destroying a component

A component has life cycle events we can use to hook on and perform custom logic, as we learned in *Chapter 4*, *Enabling User Experience 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 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 { AfterViewInit, Component, OnDestroy, OnInit, ViewChild }
from '@angular/core';
```

5. Add `OnDestroy` to the implemented interface list of the `ProductListComponent` class.

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. Can we do better than this? Yes, we can!

We can use a particular type of observable called `Subject`, which extends an `Observable` object as it is both an observer and an observable. It can emit values to multiple observers, whereas an `Observable` object unicasts only to one observer at a time. We have already met such an object before in *Chapter 4*, *Enabling User Experience with Components*. The `EventEmitter` class from the `@angular/core` npm package that we used in the output binding of a component is a `Subject`. Other cases that a `Subject` can be used for are the following:

- To pass data between components using observables
- To implement a mechanism with *search as you type* features

We will explore the way of unsubscribing from observables using a `Subject` in the product view component:

1. Open the `product-view.component.ts` file and add the following `import` statement:

```
import { Subject, takeUntil } from 'rxjs';
```

   The `takeUntil` artifact is an RxJS operator that unsubscribes from an observable when it completes. The `Subject` artifact is also part of the `rxjs` npm package.

2. Create a `productSub` property in the `ProductViewComponent` class and initialize it with an instance of the `Subject` class:

```
private productSub = new Subject<void>();
```

3. Modify the `getProduct` method to use the `takeUntil` operator:

```
private getProduct() {
  this.productviewService.getProduct(this.id).pipe(
    takeUntil(this.productSub)
```

```
  ).subscribe(product => {
    if (product) {
      this.name = product.name;
    }
  });
}
```

In the preceding method, we use the `pipe` operator to chain the `takeUntil` operator with the subscription from the `getProduct` method of the `ProductViewService` class. The `takeUntil` operator accepts a parameter of the subscription that waits for completion.

4. Import the `OnDestroy` lifecycle hook from the `@angular/core` npm package:

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

5. Add `OnDestroy` to the implemented interface list of the `ProductViewComponent` class.
6. Implement the `ngOnDestroy` method so that it completes the `productSub`:

```
ngOnDestroy(): void {
  this.productSub.next();
  this.productSub.complete();
}
```

Before completing a subject, we must call its `next` method to emit any last values to its subscribers.

That's it! We have now converted our subscription in a more declarative way that is more readable. But 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 an Angular built-in pipe that is used in conjunction with observables, and its role is two-fold. It helps us to type less code, and it 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 import the `Observable` artifact from the `rxjs` npm package:

```
import { Subscription, Observable } from 'rxjs';
```

2. Convert the `products` component property to an observable:

```
products$: Observable<Product[]> | undefined;
```

3. 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.

Feel free to remove any unused code related to the old `productsSub` property to improve the component furthermore.

4. Open the `product-list.component.html` file and modify the unordered list element to use the `async` pipe:

```
<h2>Product List</h2>
<ul>
  <li
    *ngFor="let product of (products$ | async)! | sort; let i=index"
    (click)="selectedProduct = product">
      <app-product-view [id]="i"></app-product-view>
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

In the preceding template, we use the **non-null assertion operator** to type-cast the result of passing the `products$` observable through the `async` pipe. The non-null assertion operator tricks the TypeScript compiler into ignoring `null` and `undefined` values in template expressions. In strict mode, the `async` pipe assumes that the observable passed through it *may* be `null`, which comes against our declaration of the `products$` property.

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.

Summary

It takes much more than a single chapter to cover in detail all the great things we can do with the RxJS library. 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 such as observables to interact with data streams. We explored the RxJS library and how we can use some of its 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 assorted strategies to choose from, ranging from promises to observables. You can leverage the incredible functionalities of the reactive operators 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. Now that we know how to consume asynchronous data in our components, let's discover how we can benefit from the power of the RxJS library when we want to communicate over HTTP. In the next chapter, we will learn how to use the Angular built-in 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 built-in 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 possibilities to describe what you can do 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-Fourth-Edition`.

Communicating data over HTTP

Before we dive deeper into describing the Angular built-in 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 **fetch API**. It contains all the necessary methods to connect with a server and start exchanging 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 occured');
    }
  });
```

Although the fetch API is promise-based, the promise that returns is not rejected in case of error. Instead, the request is unsuccessful when the `ok` property is not present in the `response` object. If the request to the remote URL completes, 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 details 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 more flexible for managing asynchronous operations. You are probably wondering how we can apply this pattern to an asynchronous scenario, such as consuming information from an HTTP service. You have so far become 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 digest the response better.

Let's try to convert the previous example with the fetch API to an observable. We use the `Observable` class constructor that we learned to wrap the `fetch` call in an observable stream. We replace the `log` method of the console 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 occured');
      }
    });
});
```

In the preceding snippet, the `next` method emits the response data back to subscribers as soon as they arrive. The `complete` method notifies them that no other data will be available in the stream. In the case of an error, the `error` method alerts subscribers that an error has occurred.

That's it! You have now built your 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 take any asynchronous API and turn it into an observable one 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 built-in 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. To start using it, we need to import `HttpClientModule` in the main application module, `app.module.ts`:

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

```
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

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

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

The `HttpClientModule` class provides 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 various HTTP methods. Its implementation was considered carefully to ensure that developers feel at ease while developing solutions that take advantage of this class.

In a nutshell, instances of the `HttpClient` service have access to various methods to perform common HTTP request operations, such as `GET`, `POST`, `PUT`, and every existing HTTP verb. In this book, we are interested in the most basic ones:

- `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 operations constitute the primary operations for **Create Read Update Delete (CRUD)** applications. All the previous 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. 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. There are cases, though, where we do not have access to a real backend API:

- We may work remotely, and the server is only accessible through a VPN connection to which we do not have access.
- We want to set up a quick prototype for demo purposes.
- Available HTTP endpoints are not yet ready for consumption from the backend development team, a common problem when working in a large team of different types of developers.

To overcome all the previous obstacles during development, we can use a fake server such as the **Fake Store API**. 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 that are 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.

All operations that modify data in the previous endpoints do not persist data in the database. However, they return an indication if the operation was successful. All `GET` operations in the previous endpoints return a predefined collection of data.

In this chapter, we will work only with the products and login endpoints. However, we will revisit the `cart` endpoint later in the book.

You can read more about the service at `https://fakestoreapi.com`.

Handling CRUD data in Angular

CRUD applications are widespread in the Angular world, particularly in the enterprise sector. You will hardly find any web application that does not follow this pattern. Angular does a great job 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. In particular, we will create an Angular CRUD application to manage `products` from our products module in *Chapter 7*, *Being Reactive Using Observables and RxJS*. Let's get started by scaffolding our application:

1. Create a new Angular application using the following Angular CLI command:

```
ng new my-app --defaults
```

2. Copy the global CSS styles from the code samples of *Chapter 6*, *Managing Complex Tasks with Services*, inside the `styles.css` file of the current Angular project.
3. Copy the `products` folder from *Chapter 7*, *Being Reactive using Observables and RxJS*, in the `src\app` folder of the current Angular project.
4. Open the `product-list.component.ts` file and remove the `providers` property from the `@Component` decorator. The application root injector already provides the `ProductsService` class.
5. Open the `product-list.component.html` file and modify the unordered list element so that it does not use the `<app-product-view>` component:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of (products$ | async)! | sort"
(click)="selectedProduct = product">
    {{product.name}}
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

6. Open the `app.module.ts` file and import the `products` module:

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

import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';

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

7. Open the `app.component.html` file and replace its content with the following HTML template:

```
<app-product-list></app-product-list>
```

If we run the application using the `ng serve` command, we should see the product list displayed on the page correctly:

Figure 8.1: Product list

As we can see from the previous image, our application still displays static data. We need to modify it so that it gets product data from the Fake Store API.

Fetching data through HTTP

The product list component uses the `products` service to fetch and display product data. Data are 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.module.ts` file and import the Angular module of the HTTP client:

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

import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';

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

2. Now, open the `products.service.ts` file and import the `HttpClient` class and the `map` RxJS operator:

```
import { Injectable } from '@angular/core';
import { map, Observable, of } from 'rxjs';
import { HttpClient } from '@angular/common/http';
import { Product } from './product';
```

3. Create the following interface after the `import` statements:

```
interface ProductDTO {
  title: string;
  price: number;
}
```

The preceding interface conforms to the product object model of the Fake Store API. It will act as an intermediate for transforming the response object from the API into our `Product` interface. The `Product` interface defined in the `product.ts` file contains a `name` property instead of a `title` that the API requires. The suffix `DTO` comes from **Data Transfer Object (DTO)** and is often used to indicate that the interface is part of the backend API specification model.

Collaborating with the backend team that develops the API early in a project and agreeing on the structure of the models you exchange will avoid creating intermediate interfaces.

4. Create the following property in the `ProductsService` class that represents the API products endpoint:

```
private productsUrl = 'https://fakestoreapi.com/products';
```

5. Inject `HttpClient` in the `constructor` of the `ProductsService` class:

```
constructor(private http: HttpClient) { }
```

6. Modify the `getProducts` method so that it uses the `HttpClient` service to get the list of products:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return {
        name: product.title,
        price: product.price
      }
    }))
  );
}
```

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 `ProductDTO[]` as a generic type in the `get` method to indicate that the response from the API contains a list of `ProductDTO` objects. To transform the response from the API into a list of `Product` objects that our components understand, we use the `map` operator of the RxJS library.

If we run the application using the `ng serve` command, we should see a list of products from the API similar to the following:

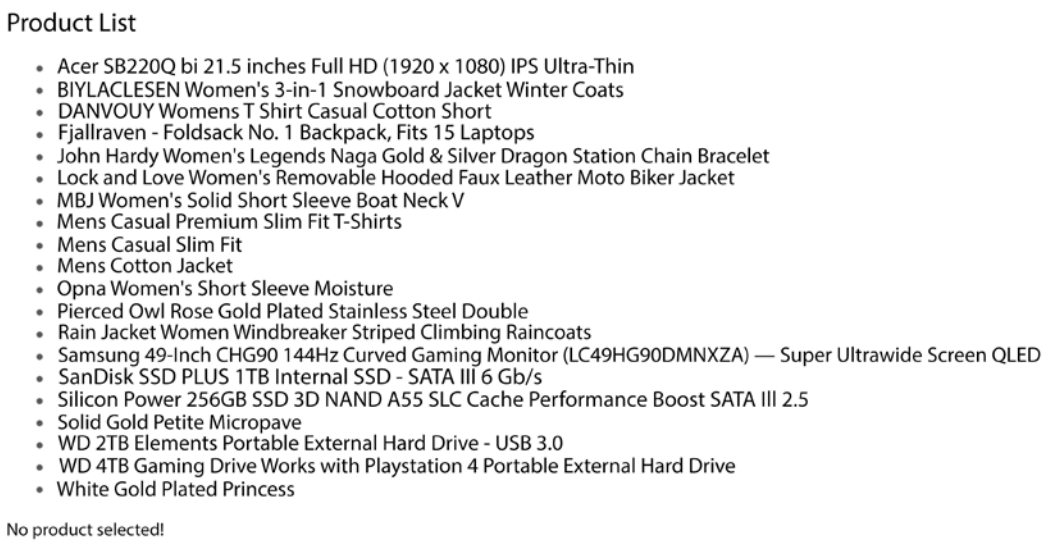


Figure 8.2: Product list from the Fake Store API

You can now remove the `products` property of the `ProductsService` class.

If you click on a product from the list, you will notice that the product details are shown correctly:

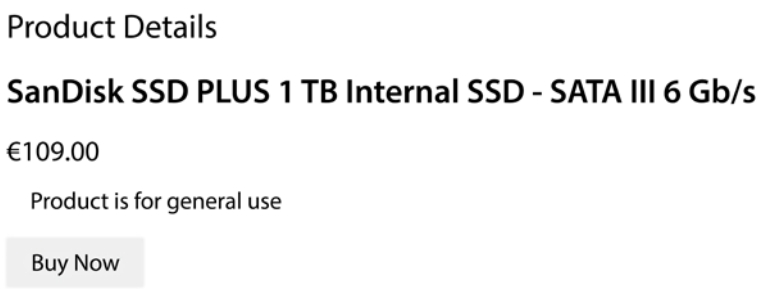


Figure 8.3: Product details

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
  *ngIf="selectedProduct; else noProduct"
  [product]="selectedProduct"
  (bought)="onBuy()">
</app-product-detail>
```

We will change the previous behavior and get the product details directly from the API using an HTTP GET request:

1. The Fake Store API contains an endpoint method that we can use to get details for a specific product based on its ID. We do not currently have an ID property in our product model, so first, let's add one in the `product.ts` file:

   ```
   export interface Product {
     id: number;
     name: string;
     price: number;
   }
   ```

2. Open the `products.service.ts` file and modify the `ProductDTO` interface to include the `id` property:

   ```
   interface ProductDTO {
     id: number;
     title: string;
     price: number;
   }
   ```

3. Create the following helper method to transform a `ProductDTO` object into a `Product` object because we will need it often inside the service:

```
private convertToProduct(product: ProductDTO): Product {
  return {
    id: product.id,
    name: product.title,
    price: product.price
  };
}
```

4. Refactor the `getProducts` method to use the newly created `convertToProduct` helper method:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    }))
  );
}
```

5. 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<ProductDTO>('${this.productsUrl}/${id}').
pipe(
    map(product => this.convertToProduct(product))
  )
}
```

   The preceding method uses the `get` method of the `HttpClient` service. It accepts the `products` endpoint followed by the product `id` as a parameter. It also defines the `ProductDTO` interface as generic because the result from the backend API will be a `ProductDTO` object.

We need to adjust the product detail component to get product details through the previous method instead of the input binding. We will still use input binding for passing the product `id` from the product list to the product detail component:

1. Open the `product-detail.component.ts` file and import the `Observable` and `ProductsService` artifacts:

```
import { Observable } from 'rxjs';
import { ProductsService } from '../products.service';
```

2. Create the following properties in the `ProductDetailComponent` class:

```
@Input() id = -1;
product$: Observable<Product> | undefined;
```

The `id` component property will be used to pass the `id` of the selected product from the list. The `product$` property will be used to call the `getProduct` method from the service.

3. Inject `ProductsService` in the `constructor` of the `ProductDetailComponent` class:

```
constructor(private productService: ProductsService) { }
```

4. Modify the `ngOnChanges` method as follows:

```
ngOnChanges(): void {
  this.product$ = this.productService.getProduct(this.id);
}
```

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. We do not want to subscribe to the `getProduct` observable because we will use the `async` pipe in the component template that will do it for us automatically.

5. Open the `product-detail.component.html` file and replace its content with the following HTML template:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <p>
    <button (click)="buy()">Buy Now</button>
  </p>
</div>
```

In the preceding template, we have removed the `NgSwitch` directive and used the `async` pipe inside the `*ngIf` directive. The `async` pipe subscribes to the `product$` observable and saves the result in the `product` template variable.

6. Finally, open the `product-list.component.html` and bind the `id` of the `selectedProduct` property to the `id` input binding of the `app-product-detail` component:

```
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (bought)="onBuy()">
</app-product-detail>
```

If we run the application using the `ng serve` command and select a product from the list, we will get an output like the following:

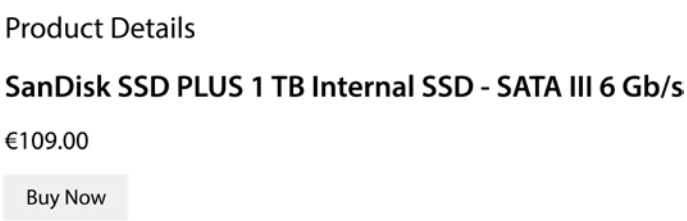


Figure 8.4: Product details

We have already 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 ones 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 `HttpClient` service, 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 for deleting an existing product.

We will start with the component for adding new products.

Adding new products

To add a new product through our application, we need to send the name and price of a new product into the Fake Store API. Before implementing the required functionality for adding products, we must refactor the `ProductListComponent` class in the `product-list.component.ts` file as follows:

```
export class ProductListComponent implements OnInit {

  selectedProduct: Product | undefined;
  products: Product[] = [];

  constructor(private productService: ProductsService) {}

  ngOnInit(): void {
    this.getProducts();
  }

  onBuy() {
    window.alert('You just bought ${this.selectedProduct?.name}!');
  }

  private getProducts() {
    this.productService.getProducts().subscribe(products => {
      this.products = products;
    });
  }

}
```

You may ask yourself why we are doing this. The `async` pipe is best suited for read-only data. In this case, we want to modify the product list by adding a new product or removing an existing one later. Plain arrays are best suited in this scenario because we can easily manipulate them using standard array functions.

You may also notice that we no longer unsubscribe from the `getProducts` observable when the component is destroyed. The Angular built-in HTTP client does a great job of unsubscribing automatically in *most* cases. However, it is advisable to always unsubscribe from observables in your components.

We must also change the `product-list.component.html` file so that the unordered list element iterates over the `products` array:

```
<h2>Product List</h2>
<ul>
  <li *ngFor="let product of products | sort" (click)="selectedProduct =
product">
    {{product.name}}
  </li>
</ul>
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (bought)="onBuy()">
</app-product-detail>
<ng-template #noProduct>
  <p>No product selected!</p>
</ng-template>
```

We can now start building the feature for adding new products to our application:

1. Open the `products.service.ts` file and add the following `addProduct` method:

```
addProduct(name: string, price: number): Observable<Product> {
  return this.http.post<ProductDTO>(this.productsUrl, {
    title: name,
    price: price
  }).pipe(
    map(product => this.convertToProduct(product))
  );
}
```

   In the preceding method, we use the `post` method of the `HttpClient` class and pass the `products` endpoint of the API, along with the details of a new product as parameters. The generic type defined in the `post` method indicates that the returned product from the API is a `ProductDTO` object. We transform the returned product into a `Product` type using the `convertToProduct` method.

2. Run the following Angular CLI command inside the `src\app\products` folder to create a new component:

```
ng generate component product-create
```

3. Open the `product-create.component.ts` file and import the following artifacts:

```
import { Component, EventEmitter, Output } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```

4. Add the following property in the `ProductCreateComponent` class using the `@Output` decorator:

```
@Output() added = new EventEmitter<Product>();
```

We will use the preceding property to emit an event back to the product list component containing the new product we created.

5. Inject `ProductsService` into the `constructor` of the component class:

```
constructor(private productsService: ProductsService) {}
```

6. Add the following `createProduct` method in the component class:

```
createProduct(name: string, price: number) {
  this.productsService.addProduct(name, price).subscribe(product =>
{
    this.added.emit(product);
  });
}
```

The preceding method accepts the `name` and `price` of a product as parameters. It calls the `addProduct` method of the `ProductsService` class and then emits the newly created product using the `added` output event.

7. Open the `product-create.component.html` file and replace its content with the following HTML template:

```
<div>
  <label for="name">Name</label>
  <input id="name" #name />
</div>
<div>
```

```
    <label for="price">Price</label>
    <input id="price" #price />
  </div>
  <div>
    <button (click)="createProduct(name.value, price.
  valueAsNumber)">Create</button>
  </div>
```

In the preceding template, we bind the `createProduct` method to the `click` event of the `Create` button. We pass the value of each `<input>` element using the respective template reference variables, `name`, and `price`. The value of the `price` variable is converted to a number using the `valueAsNumber` property because it is in string format by default.

8. Open the `styles.css` file and add the following CSS styles to give a nice look and feel to our new component:

```
input {
  font-size: 14px;
  border-radius: 4px;
  padding: 8px;
  margin-bottom: 16px;
  border: 1px solid #BDBDBD;
}

label {
  font-size: 12px;
  font-weight: bold;
  margin-bottom: 4px;
  display: block;
}
```

We have already created the component for adding new products. All we have to do now is connect it with the product list component so that when a new product is created, it is also added to the product list:

1. Open the `product-list.component.ts` file and create the following method in the `ProductListComponent` class:

```
onAdd(product: Product) {
  this.products.push(product);
}
```

The preceding method adds a new product to the list. It will be called when a new product is added through the product create component.

2. Open the `product-list.component.html` file and add the following HTML content:

```
<app-product-create (added)="onAdd($event)"></app-product-create>
```

In the preceding snippet, we add the product create component for adding new products and bind the `onAdd` component method to the `added` event. The `$event` object represents the newly added product.

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:

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.

Do not try to select a new product from the list. Remember that any new data sent to the Fake Store API are not persisted in the database.

The next feature we will add to our CRUD 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<void> {
```

```
  return this.http.patch<void>('${this.productsUrl}/${id}', { price
});
}
```

In the preceding method, we use the `patch` method of the `HttpClient` service to send the details of the product that we want to modify to the API. Alternatively, we could use 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. We do not want to update the product name in this case, so we use the `patch` method. Both methods accept the API endpoint and the object we want to update as parameters.

The return type of the `updateProduct` 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.

2. Open the `product-detail.component.ts` file and add the following method:

```
changePrice(product: Product, price: number) {
  this.productService.updateProduct(product.id, price).subscribe(()
=> {
    alert('The price of ${product.name} was changed!');
  });
}
```

The preceding method accepts an existing product and its new price as parameters. It calls the `updateProduct` method of the `ProductsService` class and displays an alert message if the product is updated successfully.

3. Open the `product-detail.component.html` file and add an `<input>` and a `<button>` element after the `<span>` element:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.
valueAsNumber)">Change</button>
  <p>
    <button (click)="buy()">Buy Now</button>
  </p>
</div>
```

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 numeric value of the `price` variable.

4. Finally, open the `product-detail.component.css` file and add the following CSS styles:

```
input {
  margin-left: 5px;
}
```

The preceding styles will give space between the `<span>` element that displays the current price and the `<input>` element that accepts the new one.

5. Run the `ng serve` command to start the Angular application and select a product from the list. The product detail component should look like the following:

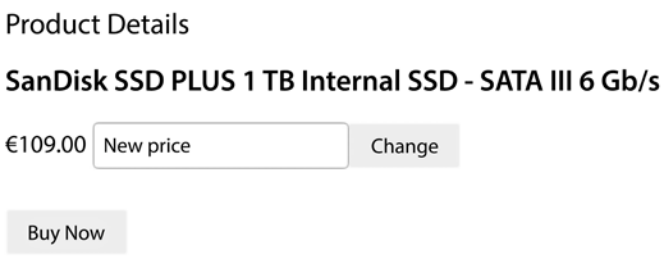


Figure 8.6: Product details

6. Enter a price in the **New price** input box and click the **Change** button. You should see an alert dialog containing the product name and stating that the price has changed:

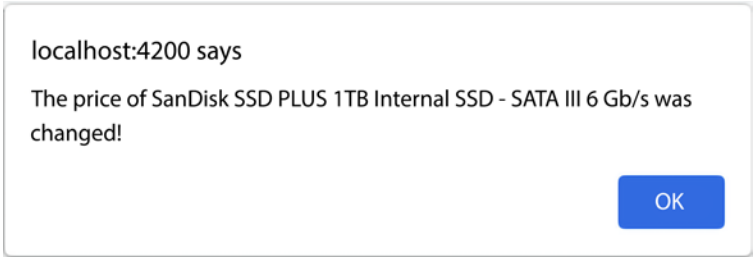


Figure 8.7: Change price alert

We can now modify a product by changing its price.

If you select the product again from the list, you will notice that the price has not been updated. Remember that any modifications to existing data sent to the Fake Store API are not persisted in the database.

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 the functionality in case users enter wrong or invalid data and want to delete them afterward. In our application, deleting an existing product will be available from the product details component:

1. Open the `products.service.ts` file and add the following method:

```
deleteProduct(id: number): Observable<void> {
  return this.http.delete<void>('${this.productsUrl}/${id}');
}
```

   The preceding method uses the `delete` method of the `HttpClient` service and passes the `products` endpoint, together with the product `id` we want to delete in the API.

   Similarly to the `updateProduct` method, we are not interested in the result of the API call. So, the signature of the method indicates that it returns `Observable<void>` as a type.

2. Open the `product-detail.component.ts` file and create a new output property in the `ProductDetailComponent` class:

```
@Output() deleted = new EventEmitter();
```

   The preceding property will be used to notify the product list component that the selected product has been deleted. The product list component will then be responsible for removing the product from the list.

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

```
remove(product: Product) {
  this.productService.deleteProduct(product.id).subscribe(() => {
    this.deleted.emit();
  });
}
```

The preceding method calls the `deleteProduct` method of the `ProductsService` class and emits the `deleted` output event.

4. Open the `product-detail.component.html` file, create a `<button>` element, and bind its `click` event to the `remove` component method:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.
valueAsNumber)">Change</button>
  <p>
    <button (click)="buy()">Buy Now</button>
    <button class="delete" (click)="remove(product)">Delete</button>
  </p>
</div>
```

5. Add an appropriate style for the new button in the `product-detail.component.css` file:

```
.delete {
  background-color: lightcoral;
  color: white;
  margin-left: 5px;
}
```

6. Open the `product-list.component.ts` file and add the following method:

```
onDelete() {
  this.products = this.products.filter(product => product !== this.
selectedProduct);
  this.selectedProduct = undefined;
}
```

The preceding method will be called when the product detail component indicates that the selected product was deleted. It filters out the deleted product from the list and sets the `selectedProduct` to `undefined` so that there is no selected product on the page.

7. Open the `product-list.component.html` file and bind the `onDelete` component method to the `deleted` event of the `app-product-detail` component:

```
<app-product-detail
  *ngIf="selectedProduct; else noProduct"
  [id]="selectedProduct.id"
  (deleted)="onDelete()"
  (bought)="onBuy()">
</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:

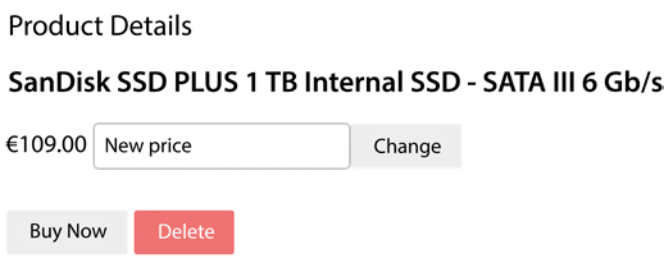


Figure 8.8: Product details

The product detail component now has a **Delete** button. When we click the button, the application deletes the product from the backend and hides it from the list.

If you reload your browser, you will notice that the product is still displayed on the list. Remember that any modifications to existing data sent to the Fake Store API are not persisted in the database.

The e-shop application we have built so far provides a **Buy Now** button that we can use to add a product to a cart. The button does not do much yet, but we will implement the full cart functionality in the following chapters. However, we should ensure that the feature will only be available to authenticated users.

In an Angular enterprise application, the product management feature must also be protected from unauthorized users. In this case, we would implement a more granular authorization scheme with user roles, where administrators would only be allowed to change and add products. We will not implement this feature, but we encourage you to experiment by yourselves.

The following section will teach us how to perform authentication and authorization in an Angular application.

Authentication and authorization with HTTP

The Fake Store API provides an endpoint for authenticating users with a username and a password. 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.

The username and password are provided by a predefined pool of users at `https://fakestoreapi.com/users`.

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 how to authenticate with the Fake Store API.

Authenticating with backend API

In Angular real-world applications, we usually consider authentication as an application feature. So, we will create a new Angular module that will handle authentication in our application. The module will contain an Angular component, allowing the user to log in to and log out of the application. An Angular service will be responsible for communicating with the Fake Store API and handling all authentication tasks.

Let's get started by setting up the new Angular module:

1. Run the following command to create a new `auth` module:

```
ng generate module auth
```

2. Navigate inside the `src\app\auth` folder and run the following command to create a new Angular service:

```
ng generate service auth
```

3. Open the `auth.service.ts` file and add the following `import` statements:

```
import { HttpClient } from '@angular/common/http';
import { Observable, tap } from 'rxjs';
```

The `tap` artifact is an RxJS operator we use when we want to handle the emitted data from an observable without modifying it.

4. Create a `private` property for the authentication token in the `AuthService` class and inject the `HttpClient` service in the `constructor`:

```
export class AuthService {

  private token = '';

  constructor(private http: HttpClient) { }

}
```

We mark the property as `private` because we do not want sensitive data to be accessible outside the `AuthService` class.

5. Create a `login` method to allow users to log in to the Fake Store API:

```
login(): Observable<string> {
  return this.http.post<string>('https://fakestoreapi.com/auth/
login', {
    username: 'david_r',
    password: '3478*#54'
  }).pipe(tap(token => this.token = token));
}
```

The preceding method initiates a POST request to the API, using the `login` endpoint and passing predefined values for `username` and `password`. The observable returned from the POST request is passed to the `tap` operator, which sets the token received from the API to the `token` service property.

6. Create a `logout` method that resets the `token` property:

```
logout() {
  this.token = '';
}
```

We have already set up the business logic for authenticating users in our Angular application. In the following section, we will learn how to start using it and control authorization access in the application.

Authorizing user access

First, we will create an authentication component that will allow our users to log in to and log out of the application:

1. Run the following command inside the `src\app\auth` folder to create a new Angular component:

```
ng generate component auth -export
```

   The preceding Angular CLI command will create the `auth` component and export it from the `auth` module so that other Angular modules can use it.

2. Open the `auth.component.ts` file and inject `AuthService`:

```
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';

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

  constructor(public authService: AuthService) { }

}
```

   In the preceding scenario, we use the `public` access modifier to inject `AuthService` because we want it to be accessible from the component template.

Although we can use an Angular service directly in a component template, limiting its content inside the component class is considered a best practice. In this scenario, we follow the former for simplicity.

3. Open the `auth.component.html` file and replace its content with the following HTML template:

```
<button
  [hidden]="authService.isLoggedIn"
  (click)="authService.login().subscribe()"
>Login</button>
<button
  [hidden]="!authService.isLoggedIn"
  (click)="authService.logout()"
>Logout</button>
```

The preceding template contains two `<button>` elements for login/logout purposes. Each button is displayed conditionally according to the value of the `isLoggedIn` property of the `AuthService` class.

4. Open the `auth.service.ts` file and create the `isLoggedIn` getter property:

```
get isLoggedIn() { return this.token !== ''; }
```

According to the preceding property, a user is considered authenticated when a token is set in the application.

5. Open the `product-detail.component.ts` file and import the `AuthService` artifact:

```
import { AuthService } from '../../auth/auth.service';
```

6. Inject `AuthService` in the `constructor` of the `ProductDetailComponent`:

```
constructor(private productService: ProductsService, public
authService: AuthService) { }
```

7. Open the `product-detail.component.html` file and use the `ngIf` directive to display the `Buy Now` button conditionally:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <input placeholder="New price" #price/>
  <button (click)="changePrice(product, price.
valueAsNumber)">Change</button>
  <p>
```

```
    <button *ngIf="authService.isLoggedIn" (click)="buy()">Buy Now</
button>
    <button class="delete" (click)="remove(product)">Delete</button>
  </p>
</div>
```

In the preceding template, we used the `ngIf` directive, not the `hidden` attribute, because we want the button to be completely removed from the DOM.

8. Import the `auth` module into the main application module file, `app.module.ts`:

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

import { AppComponent } from './app.component';
import { ProductsModule } from './products/products.module';
import { AuthModule } from './auth/auth.module';

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

9. Open the `app.component.html` file and add the `auth` component to the template:

```
<app-auth></app-auth>
<app-product-list></app-product-list>
```

Congratulations! You have gone a long way in adding 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, consider that 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` artifact from the `@angular/common/http` namespace and modifying our methods accordingly:

```
getProducts(): Observable<Product[]> {
  const options = {
    headers: new HttpHeaders({ Authorization: 'myAuthToken' })
  };
  return this.http.get<ProductDTO[]>(this.productsUrl, options).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    }))
  );
}
```

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. Request options can be an HTTP header, as in our case, or even query parameters. To set a header, we use the `headers` key 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 remaining 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 built-in 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 built-in 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.

We can create an interceptor using the `generate` command of the Angular CLI. The following command will create an Angular interceptor named `auth`:

```
ng generate interceptor auth
```

Angular interceptors must be registered with an Angular module to use them. To register an interceptor with a module, import the `HTTP_INTERCEPTORS` injection token from the `@angular/common/http` namespace and add it to the `providers` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    ProductsModule,
    AuthModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
```

In the preceding snippet, we provide `AuthInterceptor` in the main application module, `AppModule`.

An HTTP interceptor must be provided in the same Angular module that imports `HttpClientModule`.

The provided object literal contains a key named `multi` that takes a boolean value. We set it to `true` to indicate that the `HTTP_INTERCEPTORS` injection token can accept multiple service instances. Using the `multi` option does not require the `providedIn` property in the `@Injectable` decorator of the service to be present. It also enables us to combine multiple interceptors, each satisfying a particular need. But how can they cooperate and play nicely altogether?

As we can see from the `auth.interceptor.ts` file, the interceptor is an Angular service that implements the `HttpInterceptor` interface:

```
import { Injectable } from '@angular/core';
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor
} from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor() {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler):
Observable<HttpEvent<unknown>> {
    return next.handle(request);
  }
}
```

It implements the `intercept` method of the `HttpInterceptor` interface that accepts the following parameters:

- `request`: An `HttpRequest` object that indicates the current request
- `next`: An `HttpHandler` object that denotes the next interceptor in the chain

The purest form of an interceptor is to delegate requests to the next interceptor using the `handle` method. Thus, it is evident that the order in which we import interceptors in our Angular module matters. In the following diagram, you can see how interceptors process HTTP requests and responses according to their import order:

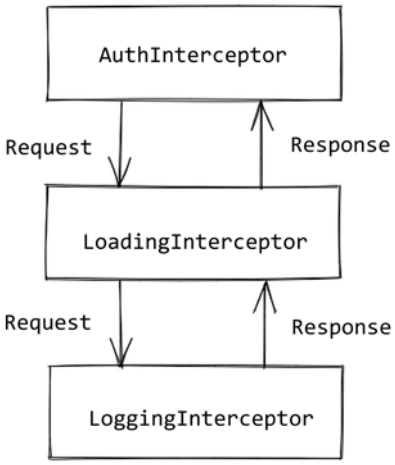


Figure 8.9: Execution order of Angular interceptors

By default, the last interceptor before sending the request to the server is a particular service named **HttpBackend**.

Now that we have covered the basics of interceptors, we will use the one we created earlier to set the authentication header for the backend API. Modify the `intercept` method of the `AuthInterceptor` class as follows:

```
intercept(request: HttpRequest<unknown>, next: HttpHandler):
Observable<HttpEvent<unknown>> {
  const authReq = request.clone({
    setHeaders: { Authorization: 'myAuthToken' }
  });
  return next.handle(authReq);
}
```

In the preceding method, 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.

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. However, the token may expire and become useless according to its configuration from the backend server.

In this case, the `auth` interceptor should take this into account and communicate with the `HttpClient` to initiate a request and get a new token. While it is tempting to inject `HttpClient` into the interceptor, it generally should be avoided unless you know what you are doing. You should be very careful because you may end up with cyclic dependencies.

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 built-in HTTP client. In this chapter, we explored the essential parts of the Angular HTTP client.

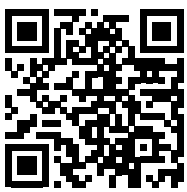
We learned how to move away from the traditional `fetch` API and use observables to communicate over HTTP. We explored the basic parts of a CRUD application using the Fake Store API as our backend. We investigated how to implement authentication and authorization in Angular applications. Finally, we learned what Angular interceptors are and how we can 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.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`



9

Navigating through Application 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 concerned ourselves with the user experience that we provide throughout the application.

Currently, our user interface is bloated, with components scattered across a single screen. We need to provide a better navigational experience and a logical way to intuitively change the application's view. Now is the right time to incorporate routing and split the different areas of interest into pages, interconnected 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, which was built with componentization in mind, and create custom links for our components to react to.

In this chapter, we will do the following:

- Introduce the Angular router
- Create an Angular application with routing
- Create feature routing modules
- Pass parameters to routes
- Enhance navigation with advanced features

Technical requirements

The chapter contains various code samples to walk you through the concept of 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-Fourth-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:

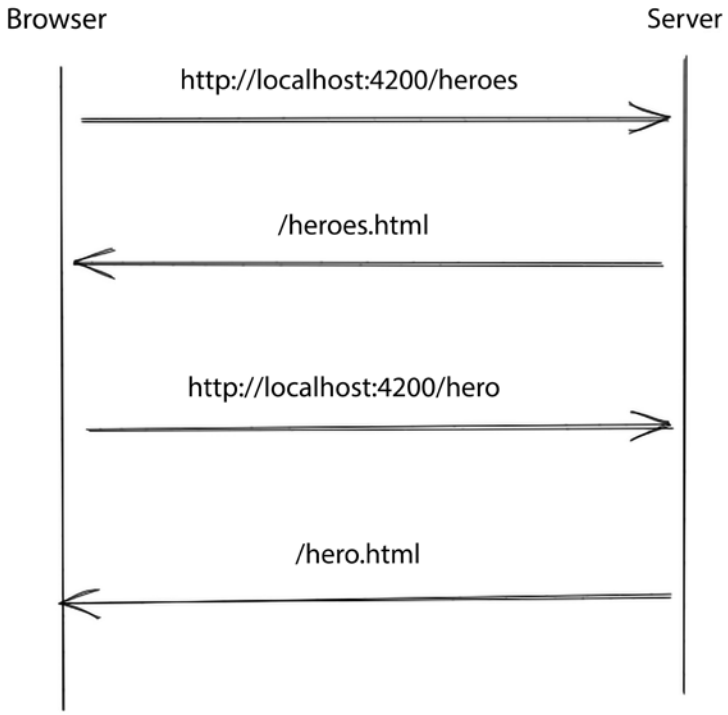


Figure 9.1: Routing in traditional web applications

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 only once during bootstrapping to get the main HTML file. Any subsequent URL changes are intercepted and handled by the router on the client. These types of applications are called **Single-Page Applications** (**SPAs**) because they do not cause a full reload of a page:

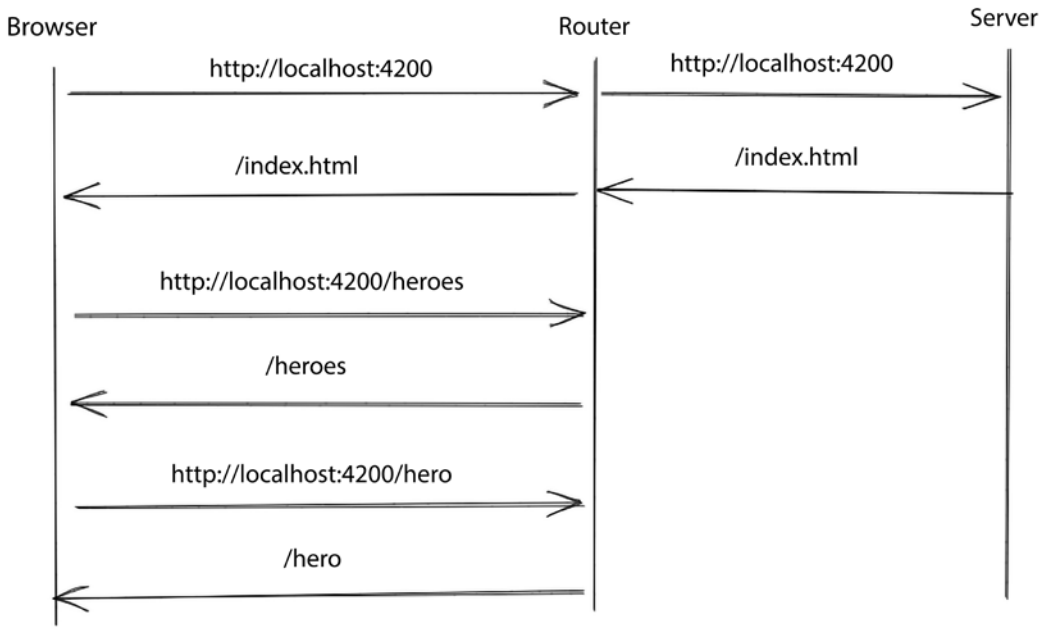


Figure 9.2: SPA architecture

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. Specify the base path for the Angular application.
2. Use an appropriate Angular module from the `@angular/router` package.
3. Configure different routes for the Angular application.
4. Decide 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 history of the browser 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 about the path it should follow when attempting to load external resources, such as media or CSS files, once it goes deeper into the URL hierarchy.

The Angular CLI automatically adds the tag when creating a new Angular application and sets the `href` value to the application root, `/`. If your application resides in a different folder than the `src\app`, you should name it after that folder.

Importing the router module

The Angular router library contains `RouterModule`, an Angular module that we need to import into our application to start using the routing features:

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

We import `RouterModule` in the main application module, `AppModule`, using the `forRoot` method:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

The `forRoot` pattern is used when a module defines services and other declarable artifacts such as components and pipes. If we try to import it normally, we will end up with multiple instances of the same service, thereby violating the singleton pattern. It works similarly to when we provide a service to the `root` injector of the application.

The `forRoot` method of `RouterModule` returns an Angular module that contains 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 route configuration of the application.

Configuring the router

The `routes` variable that we pass in the `forRoot` method is a list of `Routes` objects that specify what routes exist in the application and what components should respond to a specific route. It can look like the following:

```
const routes: Routes = [
  { path: 'products', component: ProductListComponent },
```

```
  { path: '**', component: PageNotFoundComponent }
];
```

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 either manually by changing the browser URL or by navigating using in-app links. In the first scenario, the browser will cause the application to reload, while the second will instruct the router to navigate along a route path in the application code. In our case, when the browser URL contains the `/products` path, the router creates an instance of `ProductListComponent` and displays its template on the page. On the contrary, when the application navigates to `/products` by code, the router follows the same procedure and additionally updates the URL of the browser.

If the user tries to navigate to a URL that does not match any route, Angular activates a custom type of route called a **wildcard**. The wildcard route has a `path` property with two asterisks and matches any URL. The `component` property is usually an application-specific `PageNotFoundComponent` or the main component of the application.

Rendering components

One of the directives the Angular router makes available in an Angular application is `router-outlet`. It is used as an Angular component and a placeholder for components activated with routing.

Typically, the main component of an Angular application is used only for providing the main layout of the application and orchestrating all other components. We should write it once, forget it, and not modify it when we want to add a new feature to our application. So, a typical example of the `app.component.html` file is the following:

```
<app-header></app-header>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

In the preceding HTML template, the `<app-header>` and `<app-footer>` elements are layout components. The `<router-outlet>` element is where all other components will be rendered using routing. In reality, these components are rendered as a sibling element of the `router-outlet` directive.

We have already covered the basics and provided a minimal router setup. In the next section, we will look at a more realistic example and further expand our knowledge of the routing module and how it can help us.

Creating an Angular application with routing

Whenever we have created a new Angular application through the course of this book so far, the Angular CLI has asked us whether we wanted to add routing, and we have always replied *no*. Well, it is time to respond positively and enable routing to our Angular application! In the following sections, we will put into practice all the basics that we have learned about routing:

- Scaffolding an Angular application with routing
- Adding route configuration to our Angular application
- Navigating to application routes

At the end of this section, we will have built a simple application with complete routing capabilities.

Scaffolding an Angular application with routing

We will use the Angular CLI to create a new Angular application from scratch. Run the following Angular CLI command to create a new Angular application named `my-app`:

```
ng new my-app --routing --style=css
```

The preceding command uses runtime options to skip the interactive workflow of the Angular CLI that asks questions about the application we want to build. It uses the following command-line options:

- `--routing`: Imports the Angular router to configure navigation for our application
- `--style=css`: Configures our application to use CSS for component styling

The previous Angular CLI command generates roughly the same files as usual but with one exception, the `app-routing.module.ts` file:

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

const routes: Routes = [];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

The `app-routing.module.ts` file is the routing module of the main application module. The Angular CLI names the routing module file after the actual module, appending the `-routing` suffix. It is a convention that helps us to quickly identify whether a module has routing enabled and what the respective routing module is. The name of the TypeScript class of the routing module also follows a similar convention.

The preceding file exports the `AppRoutingModule` class, an Angular module used to configure and enable the router in our application. It imports `RouterModule` using the `forRoot` method, as we have already learned in the previous section. It also re-exports `RouterModule` so that components of other modules that import `AppRoutingModule` have access to router services and directives. By default, `AppModule` imports `AppRoutingModule` in the `app.module.ts` file, so all the components of our application are enabled with routing capabilities:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

We mentioned in the previous section that `AppModule` imports `RouterModule` directly. We could have followed that approach for a minimal route configuration, but we suggest creating a separate routing module for the following reasons:

- We can change the route configuration of the application anytime, independent of the Angular module that imports it.
- We can easily test the Angular module without enabling routing. Routing is difficult to manage in unit tests.
- From the existence of a routing module, we can quickly understand that an Angular module supports routing.

Routing modules are used not only in the main application module but also in feature modules, as we will learn in the following section.

Configuring routing in our application

When we start designing the architecture of an Angular application with routing, it is easier to think about its main features as 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.

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 named `cart`:

```
ng generate component cart
```

2. Run the following command to create a new Angular component named `products`:

```
ng generate component products
```

3. Open the `app-routing.module.ts` file and add the following `import` statements:

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

4. Add two route definition objects in the `routes` variable:

```
const routes: Routes = [
  { path: 'products', component: ProductsComponent },
  { path: 'cart', component: CartComponent }
];
```

In the preceding snippet, the `products` route will activate `ProductsComponent`, and the `cart` route will activate `CartComponent`.

5. Open the `app.component.html` file and replace its content with the following HTML template:

```
<a routerLink="/products">Products</a>
<a routerLink="/cart">Cart</a>
<router-outlet></router-outlet>
```

In the preceding template, we use two router directives to perform navigation in our application, the `router-outlet` directive we have already seen and the `routerLink`. We apply the `routerLink` directive to anchor HTML elements, and we assign the route path in which we want to navigate as a value. Notice that the path should start with `/` as opposed to the `path` property in the route definition object.

6. Open the `styles.css` file and add the following CSS styles:

```
a {
  color: #1976d2;
  text-decoration: none;
  margin: 5px;
}

a:hover {
  opacity: 0.8;
}
```

We are now ready to preview our Angular application. Run the `ng serve` command and click on the **Products** link. The application should display the template of `ProductsComponent`. It should also update the URL of the browser to match the path of the route.

Now try to do the opposite. Navigate to the `root` path at `http://localhost:4200` and append the `/products` path at the end of the URL. The application should behave the same as before:

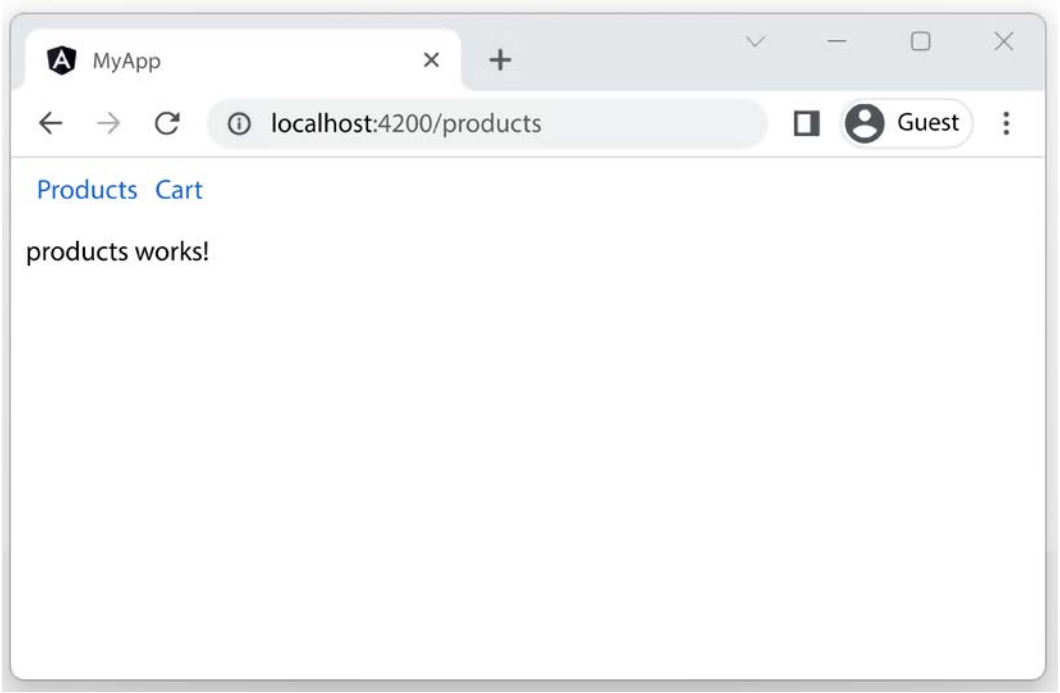


Figure 9.3: Products route

Congratulations! Your Angular application now supports in-app navigation. We have barely scratched the surface of routing in Angular. There are many router features for us to investigate waiting in the following sections. For now, let's try to move our components to a separate feature module so that we can manage it independently of the main application module.

Creating feature routing modules

At this point, we have set up the route configuration so that routing works the way it should. However, this approach doesn't scale so well. As our application grows, more and more routes may be added to the routing module of the main application module. To overcome this problem, we should create a separate feature module for our components that will also have a dedicated routing module. We already created the products module in the previous chapter. We will use it in our application so that any product-related functionality is contained inside that module:

1. Copy the global CSS styles from the code samples of *Chapter 8*, *Communicating with Data Services over HTTP*, inside the `styles.css` file of the current Angular project.
2. Delete the `src\app\products` folder from the current Angular CLI project.

3. Copy the `products` and `auth` folders from *Chapter 8*, *Communicating with Data Services over HTTP*, into the `src\app` folder of the current Angular project.
4. Remove any references to the `ProductsComponent` class from the `app.module.ts` file and import `ProductsModule` and `HttpClientModule`:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CartComponent } from './cart/cart.component';
import { ProductsModule } from './products/products.module';

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

5. Open the `app-routing.module.ts` file and modify the `products` path so that it loads `ProductListComponent`:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { CartComponent } from './cart/cart.component';
import { ProductListComponent } from './products/product-list/
product-list.component';
```

```
const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: 'cart', component: CartComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

If we run the Angular application using the `ng serve` command and click on the **Products** link, it should show the product list from the Fake Store API:

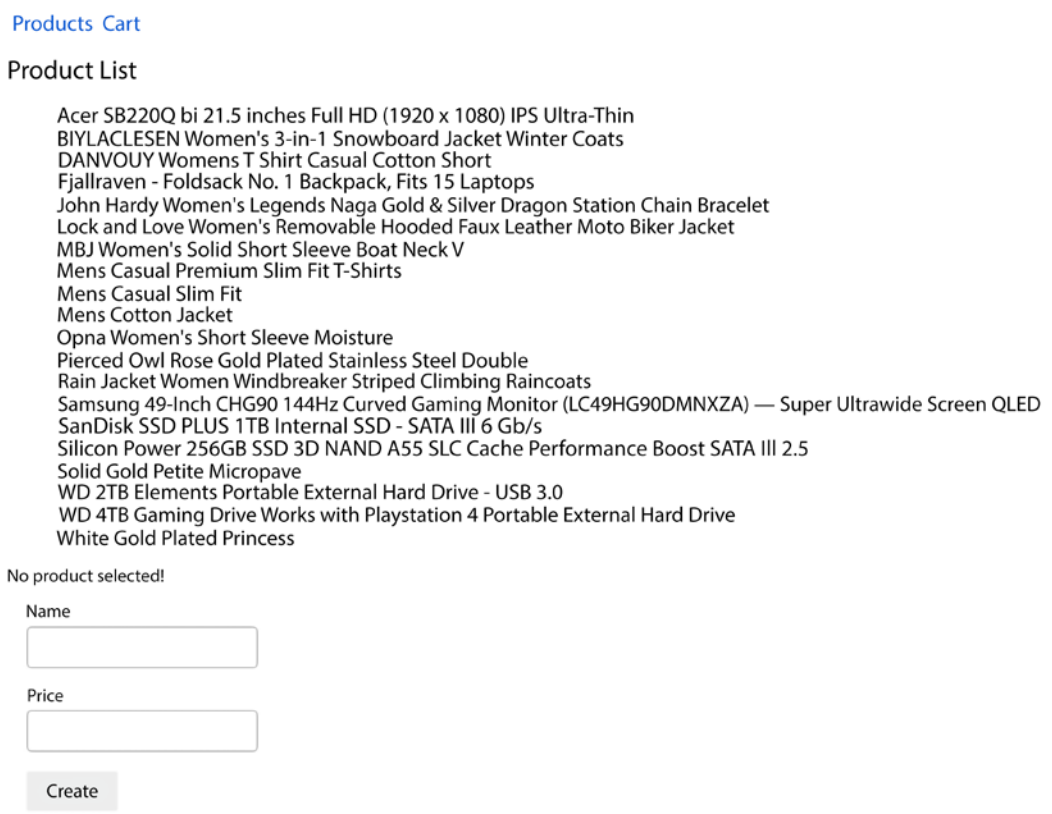


Figure 9.4: Product list

The route configuration for the `products` feature is still tied to the main application module. We will move the configuration into its own routing module inside the `src\app\products` folder:

1. Navigate to the `src\app\products` folder and create a new `products-routing.module.ts` file:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.
component';

const routes: Routes = [
  { path: 'products', component: ProductListComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

   In the previous snippet, you may have noticed that we do not import `RouterModule` using the `forRoot` method as we did before. Instead, we use the `forChild` method to import it. The `forChild` method is used when we want to register routes in a feature module. You should call the `forRoot` method *only* in the routing module of the main application module.

2. Open the `products.module.ts` file and add the following `import` statement:

```
import { ProductsRoutingModule } from './products-routing.module';
```

3. Add the `ProductsRoutingModule` class in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent,
    SortPipe,
    ProductViewComponent,
    ProductCreateComponent
  ],
```

```
  imports: [
    CommonModule,
    ProductsRoutingModule
  ],
  exports: [ProductListComponent]
})
```

4. Open the `app-routing.module.ts` file and remove the route definition object of the `products` path.
5. Open the `app.module.ts` file and change the location of `ProductsModule` in the `@NgModule` decorator so that it is imported before `AppRoutingModule`:

```
@NgModule({
  declarations: [
    AppComponent,
    CartComponent
  ],
  imports: [
    BrowserModule,
    ProductsModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

   The order that we import routing modules does matter. The router selects a route with a first-match-wins strategy. We place feature routing modules, which contain more specific routes, before the main application routing module, which contains more generic routes. Thus, we want to force the router to search through our specific route paths and then fall back to an application-specific one.

If we run the Angular application using the `ng serve` command, we will see that it is working as before. We have not introduced any new features or done anything fancy, but we have paved the way to separating our route configurations effectively. The router combines the routes of our feature module, `ProductsModule`, with those of the main application module, `AppModule`. Thus, we can continue to work with routing in our feature module without modifying the main route configuration.

Currently, the route configuration of our application is pretty straightforward. However, there are scenarios that we need to take into account when working with routing in a web application, such as the following:

- Do we want to display a specific view when we bootstrap our application?
- What will happen if we try to navigate to a non-existing route path?

In the following section, we will explore how to handle the last case so that we do not break our application.

Handling unknown route paths

We have already come across the concept of unknown routes in the *Introducing the Angular router* section. We set up a wildcard route to display `PageNotFoundComponent` when our application tries to navigate to a route path that does not exist. Now it is time to add that component for real:

1. Use the `generate` command of the Angular CLI to create a new component named `page-not-found`:

   ```
   ng generate component page-not-found
   ```

   Our application will display the newly generated component when navigating to an unknown route path.

2. Open the `page-not-found.component.html` file and replace its content with a meaningful HTML template:

   ```
   <h3>Ooops!</h3>
   <p>The requested page was not found</p>
   ```

3. Open the `app-routing.module.ts` file and add the following `import` statement:

   ```
   import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
   ```

4. Add a new route definition object in the `routes` variable. Set the `path` property to double asterisks and the `component` property to the new component that we created:

   ```
   const routes: Routes = [
     { path: 'cart', component: CartComponent },
     { path: '**', component: PageNotFoundComponent }
   ];
   ```

It is better to define the wildcard route in the `app-routing.module.ts` file. The wildcard route applies to the whole application, and thus it is not tied to a specific feature. Additionally, 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 following output:

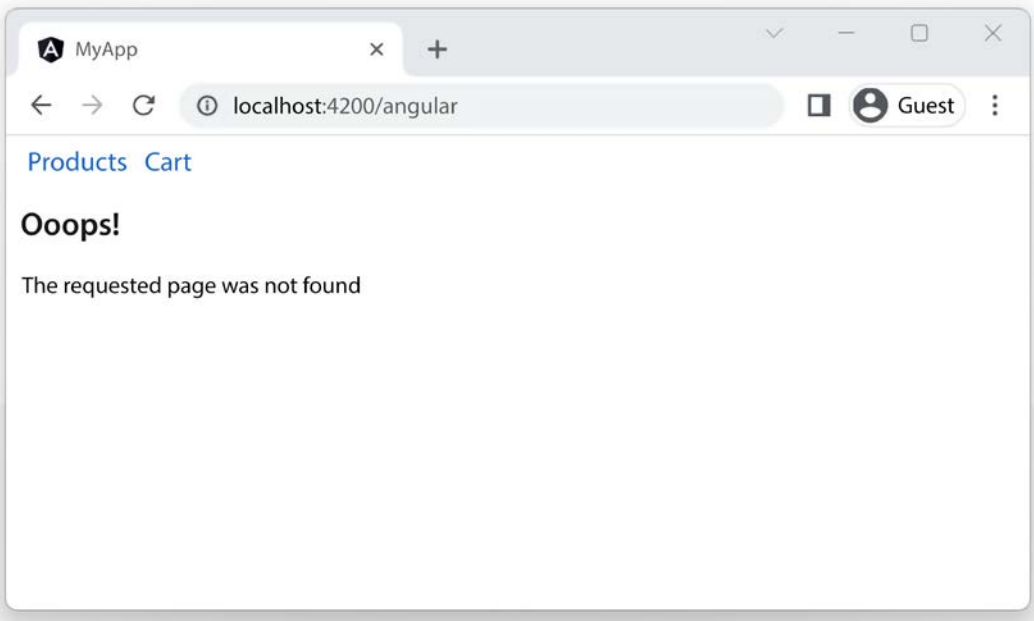


Figure 9.5: Page not found route

When the router encounters an unknown route, it navigates to the wildcard route, but the browser still points to the invalid URL.

Try to navigate to the root path of the application, `http://localhost:4200`, and you will notice that the template of `PageNotFoundComponent` is still visible on the screen. We have accidentally broken our application! How did this happen?

The `href` attribute of the `base` tag in the `index.html` file is the location at which an Angular application starts, as we learned in the *Introducing the Angular router* section.

The Angular CLI sets the value of `href` to `/` by default when creating a new Angular application. We have also learned that a route does not contain `/` in its `path` property. So, when our application bootstraps, it loads in an empty route path. According to our route configurations, we have not defined such a path. Thus, the router falls back to the wildcard route. We need to define a default route for our Angular application, which brings us to the first scenario we described: how to define a default route path when our application bootstraps.

Setting a default path

We set the `path` property of a route to an empty string to indicate that the route is the default one for an Angular application. In our case, we want the default route path to display the product list. Open the `products-routing.module.ts` file and add a new route definition object *at the end* of the existing routes:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.
component';

const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: '', redirectTo: '/products', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

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 route 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 empty route path after all other routes because, as we have already learned, the order of the routes is important. We want more specific routes before less specific ones. In the following diagram, you can see the order in which the router resolves paths in our application:

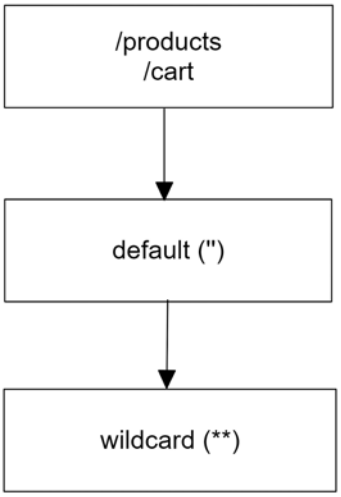


Figure 9.6: Route path resolve process

We have already learned how to navigate our application using the `routerLink` directive. It is the preferred way when using anchor elements in a component template. However, in a real-world application, we also use buttons for navigation. In the following section, we will learn how to navigate to a route path **imperatively** using a `<button>` element.

Navigating imperatively to a route

When we navigate to a wildcard route, the template of the `component` property is displayed on the screen. However, as we have seen, the address bar of the browser stays on the invalid URL. So, we need to provide a way for the user to escape from this route:

1. Open the `page-not-found.component.html` file and add a `<button>` HTML element:

```
<h3>Ooops!</h3>
<p>The requested page was not found</p>
<button (click)="goHome()">Home</button>
```

   In the preceding HTML template, we have added an event binding to the `click` event of the `<button>` element that points to the `goHome` component method, which does not exist yet.

2. Open the `page-not-found.component.ts` file and add the following `import` statement:

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

3. Inject the `Router` service in the `constructor` of the `PageNotFoundComponent` class:

```
constructor(private router: Router) { }
```

4. Add the following `goHome` method:

```
goHome() {
  this.router.navigate(['/']);
}
```

   In the preceding method, we call the `navigate` method of the `Router` service to navigate into the root path of the application. It accepts a **link parameters array** containing the destination route path we want to navigate.

It is worth noting that the link parameters array syntax can also be used with the `routerLink` directive. For example, we could have written the `app.component.html` file as follows:

```
<a [routerLink]="['/products']">Products</a>
<a [routerLink]="['/cart']">Cart</a>
<router-outlet></router-outlet>
```

It is perfectly fine to use imperative navigation with an anchor element and a `routerLink` directive with a `<button>` element. However, using them, as suggested in this book, is more semantically correct in HTML. The `routerLink` directive modifies the behavior of the target element and adds an `href` attribute, which targets only anchor elements.

Until now, we have relied on the address bar of the browser to indicate which route path is active at any given time. We could improve the user experience by using CSS styling, as we will learn in the following section.

Decorating router links with styling

The module of the Angular router exports the `routerLinkActive` directive, which we can use to change the style of a route when it is active. It works similarly to the class binding we learned about in *Chapter 4*, *Enabling User Experience with Components*. It accepts a list of class names or a single class that is added when the link is active and removed when it becomes inactive.

To use it in our Angular application, we need first to create a class for active links in the `styles.css` file:

```
.active {
  color: black;
}
```

We can then add the `routerLinkActive` directive to both links in the `app.component.html` file and set it with the `active` class name:

```
<a routerLink="/products" routerLinkActive="active">Products</a>
<a routerLink="/cart" routerLinkActive="active">Cart</a>
<router-outlet></router-outlet>
```

Now, when we click on a link in our application, its color turns to `black` to denote that the link is active.

We have already learned that we can navigate to a route with a static path value. In the next section, we will learn how to do this when the path changes dynamically using 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 details of an item. 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.

Currently, the product list component passes the `id` of the selected product via input binding. Input binding cannot be used if the component is activated with routing. We will use the Angular router to pass the product `id` as a route parameter:

1. Open the `products-routing.module.ts` file and add a new route definition:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductDetailComponent } from './product-detail/product-
detail.component';
import { ProductListComponent } from './product-list/product-list.
component';

const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: 'products/:id', component: ProductDetailComponent },
  { path: '', redirectTo: '/products', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

   The colon character denotes `id` as a route parameter in the new route definition object. If a route has more than one parameter, we separate them with `/`. The parameter name is important when we want to consume its value in our components, as we will learn later.

2. Open the `product-list.component.html` file and modify its HTML content so that it uses the new route definition:

```
<ul>
  <li *ngFor="let product of products | sort">
    <a [routerLink]="['/products', product.id]">{{product.name}}</a>
  </li>
</ul>
<app-product-create (added)="onAdd($event)"></app-product-create>
```

The preceding version of the `product-list.component.html` file is much different from the previous one because we have removed the notion of the selected product. The product list does not need to keep the selected product in its local state because we are navigating away from the list upon selecting a product.

We have added an anchor element inside the `<li>` element and attached the `routerLink` directive to it. 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 second parameter in the array. The `routerLink` directive requires property binding when dealing with dynamic routes. It will create an `href` attribute that contains the `/products` path, followed by the value of its `id` property.

3. Open the `product-detail.component.ts` file and import the `OnInit`, `ActivatedRoute`, and `switchMap` artifacts:

```
import { Component, OnInit, Input, Output, EventEmitter, OnChanges }
from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable, switchMap } from 'rxjs';
```

The Angular router exports the `ActivatedRoute` service, which we can use to retrieve information about the currently active route, including any parameters.

4. Inject the `ActivatedRoute` service into the `constructor` of the `ProductDetailComponent` class:

```
constructor(
  private productService: ProductsService,
  public authService: AuthService,
  private route: ActivatedRoute
) { }
```

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

```
export class ProductDetailComponent implements OnInit, OnChanges {
```

6. 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. We use the `switchMap` RxJS operator to pipe the `id` parameter from the `paramMap` observable to the `getProduct` method of the `ProductsService` class.

It is also worth noting the following:

- The `paramMap` observable returns an object of the `ParamMap` type. We can use the `get` method of the `ParamMap` object and pass the name of the parameter we defined in the route configuration to access its value.
- 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** page:

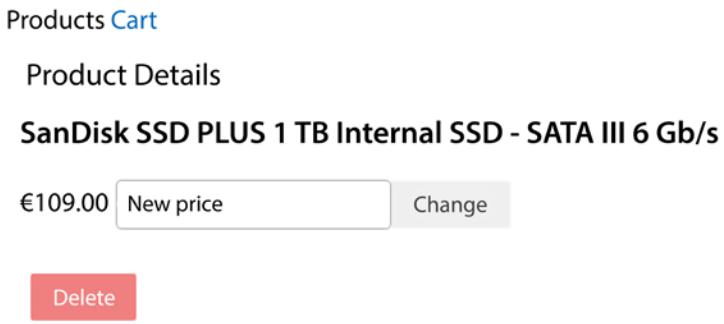


Figure 9.7: Product Details page

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. So, what's the point of using it after all?

The router can reuse 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

Using child routes is a perfect solution when we want to have a landing page component that will provide routing to other components in a feature module. It 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 the following:

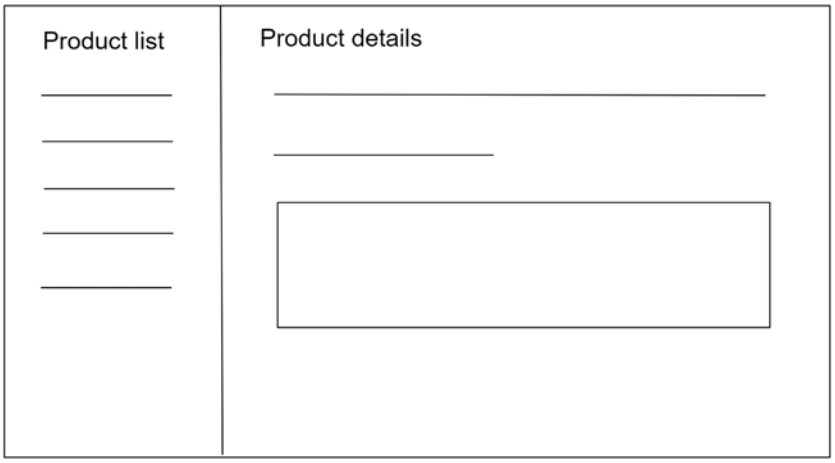


Figure 9.8: Master-detail layout

The scenario described in the previous diagram requires the product list component to contain a `<router-outlet>` element that will render the product details component when the related route is activated.

The product details component is 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 select 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 products module, in this case, would be as follows:

```
const routes: Routes = [
```

```
  {
    path: 'products',
    component: ProductListComponent,
    children: [
      { path: ':id', component: ProductDetailComponent },
    ]
  },
  { path: '', redirectTo: '/products', pathMatch: 'full' }
];
```

We use the `children` property of a route definition object to define child routes, which contain a list of route definition objects. Notice also that we removed the word `products` from the `path` property of the product route. We wanted to clarify that it is a child of the `products` route and should be accessed using the `/products/:id` path.

We must also change the `routerLink` directive of the anchor elements in the `product-list.component.html` file so that our application will work correctly:

```
<a [routerLink]="['./', product.id]">{{product.name}}</a>
```

Notice that we replaced `/products` with `./`. What is this strange syntax? It is called **relative navigation** and tells the router to navigate to a specific route relative to the current activated route. It is the opposite of the current syntax we have used so far, **absolute navigation**.

For example, the `./` path indicates to navigate relative to the current level, which is `/products`, in our case. If the route we wanted to navigate was one level above the products route, we would have used `../` as a path. You can think of it as navigating between folders using the command line. The same syntax applies to imperative navigation also:

```
this.router.navigate(['./', product.id], { relativeTo: this.route });
```

In this case, we pass an additional object after the link parameters array that defines the `relativeTo` property pointing to the current activated route.

Relative navigation is considered a better choice over absolute navigation because it is easier to refactor. It decouples hardcoded links by defining paths relative to the current route. Imagine moving a bunch of components around, and suddenly all your hardcoded paths are wrong. Navigation inside a feature module works as expected, even if you decide to change the parent route.

We have learned how to take advantage of the `paramMap` observable in Angular routing. In the following section, we will discuss an alternative approach using route snapshots.

Taking a snapshot of route parameters

Currently, in our application 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 select 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 removed from the DOM, and the product list component is added. 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 always 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, which we can access as we would access a plain object in TypeScript.

If you are sure your component will not be reused, you should use the snapshot approach since it is also more readable.

So far, we have dealt with routing parameters in the form `products/:id`. We use these parameters when we want to route to a component that requires the parameter to work correctly. In our case, the product details component requires the `id` parameter to get details of a specific product. However, there is another type of route parameter that is considered optional, as we will learn in the following section.

Filtering data using query parameters

Query parameters are considered optional because they aim to provide optional services such as sorting or filtering data. Some examples are as follows:

- `/products?sortOrder=asc`: Sorts a list of products in ascending order
- `/products?page=3&pageSize=10`: Splits a list of products into pages of 10 records and gets the third page

Query parameters are recognized in a route by the `?` character. We can combine multiple query parameters by chaining them with an ampersand (`&`) character. The `ActivatedRoute` service contains a `queryParamMap` observable that we can subscribe to in order to get query parameter values. It returns a `ParamMap` object, similar to the `paramMap` observable, which we can query to get parameter values. For example, to retrieve the value of a `sortOrder` query parameter, we would use it as follows:

```
ngOnInit(): void {
  this.route.queryParamMap.subscribe(params => {
    console.log(params.get('sortOrder'));
  });
}
```

The `queryParamMap` property is also available when working with snapshot routing to get query parameter values.

Now that we have learned how 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 our Angular applications.

Enhancing navigation with advanced features

So far, we have covered basic routing, with route parameters as well as 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 the UX
- Lazy-loading routes to speed up the response time

In the following sections, we will learn about all these techniques in more detail.

Controlling route access

When we want to prevent unauthorized access to a particular route, we use a specific Angular concept called a **guard**. An Angular guard can be of the following types:

- **canActivate**: Controls whether a route can be activated.
- **canActivateChild**: Controls access to child routes of a route.
- **canDeactivate**: Controls whether a route can be deactivated. Deactivation happens when we navigate away from a route.
- **canLoad**: Controls access to a route that loads a lazy-loaded module.
- **canMatch**: Controls access to the same route path based on application conditions.

To create a guard that will allow access to a route depending on whether the user is authenticated or not, we will run the following steps:

1. Create a file named `auth.guard.ts` inside the `src\app\auth` folder.
2. Add the following `import` statements at the top of the file:

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from './auth.service';
```

3. Create an `authGuard` function:

```
export const authGuard: CanActivateFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn) { return true; }

  return router.parseUrl('/');
};
```

In the preceding function, we use the `inject` method to inject the `AuthService` and `Router` services into the function. The `inject` method behaves the same as if we have injected both services into the constructor of a TypeScript class. We then check the value of the `isLoggedIn` property. If it is `true`, the application can navigate to the specified 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.

4. Open the `app-routing.module.ts` file and add the following `import` statement:

```
import { authGuard } from './auth/auth.guard';
```

5. Add the `authGuard` function in the `canActivate` array of the `cart` route:

```
const routes: Routes = [
  {
    path: 'cart',
    component: CartComponent,
    canActivate: [authGuard]
  },
  { path: '**', component: PageNotFoundComponent }
];
```

Only authenticated users can now access the shopping cart. If you run the application using the `ng serve` command and click on the **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 seems to be because the redirection that occurs due to the authentication guard does not have any effect when you are already in the redirected route. Select a product from the list before clicking the **Cart** link to understand how the guard works.

To be able to access the shopping cart, we need to restore the login functionality from *Chapter 8*, *Communicating with Data Services over HTTP*:

1. Open the `app.module.ts` file and add the following `import` statement:

```
import { AuthModule } from './auth/auth.module';
```

2. Add the `AuthModule` class in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent,
    CartComponent,
    PageNotFoundComponent
  ],
  imports: [
    BrowserModule,
```

```
    ProductsModule,
    AppRoutingModule,
    HttpClientModule,
    AuthModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

3. Open the `app.component.html` file and add the `<app-auth>` component at the top of the file:

```
<app-auth></app-auth>
<a routerLink="/products" routerLinkActive="active">Products</a>
<a routerLink="/cart" routerLinkActive="active">Cart</a>
<router-outlet></router-outlet>
```

If we run the application now, we can use the **Login** button to authenticate with the API and access the cart functionality.

Preventing navigation away from a route

Guards are used not only to prevent access to a route but also to prevent navigation away from it. 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:

1. Create a file named `checkout.guard.ts` inside the `src\app` folder.
2. Add the following `import` statements at the top of the file:

```
import { CanDeactivateFn } from '@angular/router';
import { CartComponent } from './cart/cart.component';
```

3. Create the following `checkoutGuard` function:

```
export const checkoutGuard: CanDeactivateFn<CartComponent> = () => {
  const confirmation = confirm('You have pending items in your cart.
Do you want to continue?');
  return confirmation;
};
```

In the preceding function, we set the type of the `CanDeactivateFn` function to `CartComponent` because we want to check whether the user navigates away from this component only.

In a real-world scenario, you may need to create a generic guard to support additional components.

We then use the global `confirm` method to display a confirmation dialog before navigating away from the cart component.

4. Open the `app-routing.module.ts` file and add the following `import` statement:

```
import { checkoutGuard } from './checkout.guard';
```

5. A route definition object contains a `canDeactivate` array similar to `canActivate`. Add the `checkoutGuard` function to the `canDeactivate` array of the `cart` route:

```
const routes: Routes = [
  {
    path: 'cart',
    component: CartComponent,
    canActivate: [authGuard],
    canDeactivate: [checkoutGuard]
  },
  { path: '**', component: PageNotFoundComponent }
];
```

For such a simple scenario, we could have written the logic of the `checkoutGuard` function inline and avoided 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 **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 the following dialog:

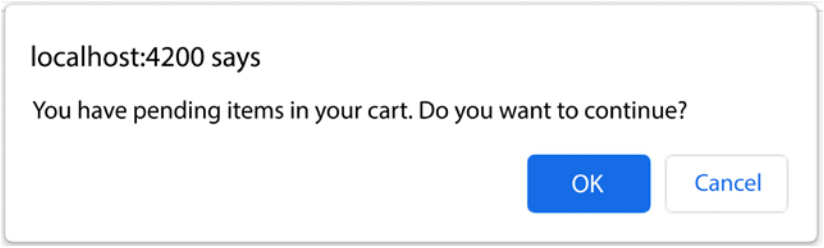


Figure 9.9: Pending items dialog

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 root path of the application.

Prefetching route data

You may have noticed that when you select a product from the list and navigate to the product details component, there is a delay in displaying the product details data. It is reasonable since we are making an HTTP request to the backend API. However, there is flickering in the user interface, which is bad for the user experience. Thankfully, the Angular router can help us to fix that. We can use a **resolver** to prefetch the details of a product so that they are available when activating the route and displaying the component.

A resolver can be handy when we want to handle possible errors before activating a route. In our case, it would be more appropriate not to navigate to the product details component if the `id` we pass as a route parameter does not exist in the backend.

Let's create a route resolver for our product details component:

1. Create a file named `product-detail.resolver.ts` inside the `src\app\products` folder.
2. Add the following `import` statements at the top of the file:

```
import { inject } from '@angular/core';
import { ActivatedRouteSnapshot, ResolveFn } from '@angular/router';
import { Product } from './product';
import { ProductsService } from './products.service';
```

3. Create the following `productDetailResolver` function:

```
export const productDetailResolver: ResolveFn<Product> = (route:
ActivatedRouteSnapshot) => {
  const productService = inject(ProductsService);

  const id = Number(route.paramMap.get('id'));
  return productService.getProduct(id);
};
```

A resolver is a function of the `ResolveFn<T>` type, where `T` is the resolved data type. It can return resolved data either synchronously or asynchronously. In our case, since we are communicating with a backend API using the HTTP client, we need to return an observable of a `Product` object.

The `productDetailResolver` function injects the `ProductsService` class, gets the value of the `id` route parameter, and converts it into a number. It then calls the `getProduct` method of the `ProductsService` class, passing the `id` as a parameter.

4. Open the `products-routing.module.ts` file and add the following `import` statement:

```
import { productDetailResolver } from './product-detail.resolver';
```

5. Add a `resolve` property to the route definition object that activates `ProductDetailComponent`:

```
const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  {
    path: 'products/:id',
    component: ProductDetailComponent,
    resolve: {
      product: productDetailResolver
    }
  },
  { path: '', redirectTo: '/products', pathMatch: 'full' }
];
```

The `resolve` property is an object that contains a unique name as a key and the TypeScript class of the resolver as a value. The name of the key is important because we will use that in our components to access the resolved data.

6. Open the `product-detail.component.ts` file and import the `of` operator from the `rxjs` npm package:

```
import { Observable, of, switchMap } from 'rxjs';
```

7. Modify the `ngOnInit` method so that it subscribes to the `data` property of the `ActivatedRoute` service:

```
ngOnInit(): void {
  this.product$ = this.route.data.pipe(
    switchMap(data => of(data['product']))
  );
}
```

   In the preceding snippet, the `data` observable emits an object where the value of the requested product exists in the `product` key of the object. Notice that we use the `switchMap` operator to return the product in a new observable.

If you run the application now, you will notice no flickering when navigating to the product details component, and data are displayed at once. However, you may notice a slight delay upon selecting the product from the list. It is the delay introduced by the HTTP request to the backend API that originates from the resolver.

Lazy-loading routes

At some point, our application may grow in size, and the amount of data we put into it may also grow. The result is that the application may take a long time to start initially, or certain parts can take a long time to start. To overcome these problems, we can use a technique called **lazy loading**.

Lazy loading means that we don't load all parts of our application initially. When we refer to parts, we mean Angular modules. Application modules can be separated into chunks that are only loaded when needed. There are many advantages of lazy loading a module in an Angular application:

- Feature modules can be loaded upon request from the user.
- Users that visit certain areas of your application can significantly benefit from this technique.
- We can add more features in a lazy-loaded module without affecting the overall application bundle size.

To understand how lazy loading in Angular works, we will create a new module with a component that displays information about our e-shop application:

1. Run the following command to create an Angular module with routing enabled:

```
ng generate module about --routing
```

2. Create a component named `about-info` inside the `src\app\about` folder:

```
ng generate component about-info
```

3. Open the `about-routing.module.ts` file and add the following `import` statement:

```
import { AboutInfoComponent } from './about-info/about-info.
component';
```

4. Add a new route definition object in the `routes` variable to activate `AboutInfoComponent`:

```
const routes: Routes = [
  { path: '', component: AboutInfoComponent }
];
```

   In the preceding snippet, we set the `path` property to an empty string so that the route is activated by default.

5. Add a new anchor element to the `app.component.html` file that links to the newly created route:

```
<app-auth></app-auth>
<a routerLink="/products" routerLinkActive="active">Products</a>
<a routerLink="/cart" routerLinkActive="active">Cart</a>
<a routerLink="/about" routerLinkActive="active">About Us</a>
<router-outlet></router-outlet>
```

6. Finally, open the `app-routing.module.ts` file and add a new route definition object:

```
const routes: Routes = [
  {
    path: 'cart',
    component: CartComponent,
    canActivate: [authGuard],
    canDeactivate: [checkoutGuard]
  },
  {
```

```
    path: 'about',
    loadChildren: () => import('./about/about.module').then(m =>
m.AboutModule)
  },
  { path: '**', component: PageNotFoundComponent }
];
```

The `loadChildren` property of a route definition object is used to lazy load Angular modules. It returns an arrow function that uses a dynamic import statement to lazy load the module. The `import` function accepts the relative path of the module we want to import, returning a promise object that contains the TypeScript class of the Angular module we want to load.

We did not import `AboutModule` in the main application module. Otherwise, it would have been loaded twice, eagerly from the main application module and lazily from the `about` route.

Run the application using the `ng serve` command and open the browser's developer tools. Click the **About Us** link, and inspect the requests in the **Network** tab:

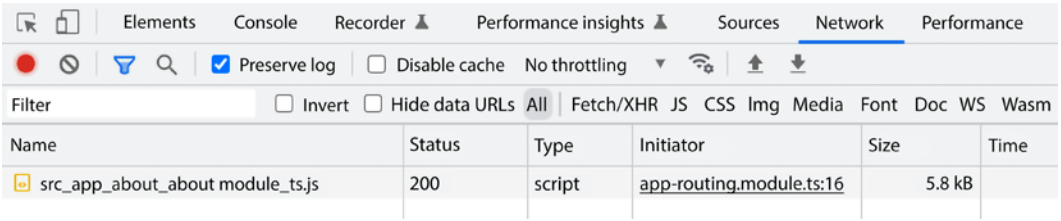


Figure 9.10: Lazy-loaded module

The application initiates a new request to the `src_app_about_about_module_ts.js` file, which is the bundle of the `about` module. The Angular framework creates a new bundle for each module that is lazy-loaded and does not include it in the main application bundle.

If you navigate away and click on the **About Us** link again, you will notice that the application does not make a new request to load the module. As soon as a lazy-loaded module is requested, it is kept in memory and can be used for subsequent requests.

A word of caution, however. As we learned in *Chapter 6*, *Managing Complex Tasks with Services*, an Angular service is registered with the root injector of the application using the `providedIn` property of the `@Injectable` decorator.

Lazy-loaded modules create a separate injector that is an immediate child of the root application injector. If you use an Angular service registered with the root application injector in a lazy-loaded module, you will end up with a separate service instance in both cases. So, we must be cautious as to how we use services in lazy-loaded modules.

Lazy-loaded modules are standard Angular modules, so we can control their access using guards.

Protecting a lazy-loaded module

We can control unauthorized access to a lazy-loaded module similar to how we can do so in eagerly loaded ones. However, our guards need to implement a different interface for this case, the `CanLoad` interface.

We will extend our authentication guard for use with lazy-loaded modules.

1. Open the `auth.guard.ts` file and import `CanLoadFn` from the `@angular/router` npm package:

```
import { CanActivateFn, CanLoadFn, Router } from '@angular/router';
```

2. Add the `CanLoadFn` type to the `authGuard` function:

```
export const authGuard: CanActivateFn | CanLoadFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isLoggedIn) { return true; }

  return router.parseUrl('/');
};
```

3. As with all previous guards, we must register the `authGuard` function with the lazy-loaded route using the `canLoad` array of the route definition object. Open the `app-routing.module.ts` file and add the `authGuard` function in the `canLoad` array of the `about` route:

```
{
  path: 'about',
  loadChildren: () => import('./about/about.module').then(m =>
m.AboutModule),
  canLoad: [authGuard]
}
```

If we now run the application and click on the **About Us** link, we will notice that we cannot navigate to the **About** page unless we are authenticated.

We have already learned about standalone components in *Chapter 4*, *Enabling User Experience with Components*. Standalone components are not registered with an Angular module but can be lazy loaded, as we will see in the following section.

Lazy loading components

We have already learned how to create standalone components, pipes, and directives in Angular applications. Before the standalone option, it was cumbersome and complicated to load a component dynamically. It required many lines of code and advanced Angular techniques from the developer's perspective. However, with the introduction of standalone APIs from Angular, developers now have a powerful tool at their disposal to satisfy even the most complex business needs in their Angular applications.

The Angular router can lazy load not only Angular modules but also standalone components. The route definition object contains a `loadComponent` property, similar to `loadChildren` for modules, allowing us to pass an Angular component for lazy loading. We will learn more by converting our `about` component into a standalone one and lazy loading it:

1. Open the `about-info.component.ts` file and add the `standalone` property in the `@Component` decorator:

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

@Component({
  selector: 'app-about-info',
  templateUrl: './about-info.component.html',
  styleUrls: ['./about-info.component.css'],
  standalone: true
})
export class AboutInfoComponent {}
```

2. Open the `about.module.ts` file and remove any occurrences of the `AboutInfoComponent` class:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AboutRoutingModule } from './about-routing.module';
```

```
@NgModule({
  imports: [
    CommonModule,
    AboutRoutingModule
  ]
})
export class AboutModule { }
```

We need to remove it specifically from the `declarations` array of the `@NgModule` decorator. Otherwise, it will not be a standalone component.

3. Open the `app-routing.module.ts` file and modify the `about` route so that it uses the `loadComponent` property to lazily load `AboutInfoComponent`:

```
{
  path: 'about',
  loadComponent: () => import('./about/about-info/about-info.
component').then(c => c.AboutInfoComponent)
}
```

Run the application using the `ng serve` command and click the **About Us** link. The component is still loaded on the page, but a different chunk appears in the **Network** tab of the browser console:

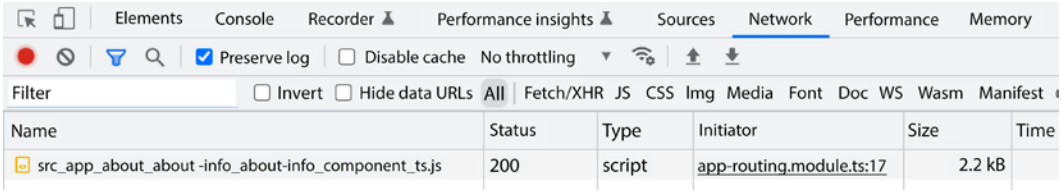


Figure 9.11: Lazy-loaded component

The application initiates a new request to the `src_app_about_about-info_about-info_component_ts.js` file this time. Everything related to lazy loading we saw in the module example also applies here.

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 when it comes to collecting data from the user. Use cases vary from allowing users to log in, filling in payment information, booking a flight, or even performing a search. Form data can later be persisted on local storage or be sent to a server using a backend API. A form usually has the following characteristics that enhance the user experience of a web application:

- Can define different kinds of input fields
- Can set up different kinds of validations and display validation errors to the user
- Can support 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 than the other; you have to go with the one that suits your scenario the best. The main difference between the two approaches is how they manage data:

- Template-driven forms are easy to set up and add to an Angular application, but they do not scale well. 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 are more robust when it comes to scaling and testing and when they are not interacting with the change detection cycle. They operate in the component class to manage input controls and set up 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.

In this chapter, we will focus mainly on reactive forms due to their wide popularity in the Angular community. More specifically, we will cover the following topics:

- Introducing forms to web apps
- Data binding with template-driven forms
- Using reactive patterns in Angular forms
- Validating controls in a reactive way
- Modifying forms dynamically
- Manipulating form data
- Watching state changes and being reactive

Technical requirements

The chapter contains various code samples to walk you through the concept of 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-Fourth-Edition`

Introducing forms to web apps

A form in a web application consists of a `<form>` HTML element that contains some `<input>` elements for entering data and a `<button>` element for handling that data. The form can retrieve data and either save it locally or send it to a server for further manipulation. The following is a simple form that is used for logging a user in to 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 clicking on the button or pressing *Enter* on any input control. We could have added another button with a **reset** type to clear 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:

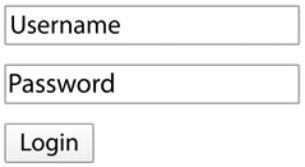


Figure 10.1: Login form

Web applications can significantly enhance the user experience by using forms that provide features such as autocomplete in input controls or prompting 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.

Data binding with template-driven forms

Template-driven forms are one of two different ways of integrating forms with Angular. It is an approach that is not widely embraced by the Angular community for the reasons described previously. Nevertheless, it can be powerful in cases where we want to create small and simple forms for our Angular application. Template-driven forms can stand out when used with the `ngModel` directive to provide two-way data binding in our components.

We learned about data binding in *Chapter 4*, *Enabling User Experience with Components*, and how we can use different types to read data from an HTML element or component and write data to it. In this case, binding is either one way or another, which is also called **one-way binding**. 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. Before we can start using Angular forms, we need to configure our Angular application by importing `FormsModule`, an appropriate built-in Angular module for working with template-driven forms:

1. Run the following command to create a new Angular application:

   ```
   ng new my-app --routing --style=css
   ```

   The preceding command will create an Angular application that enables routing and uses CSS for component styling.

2. Copy the contents of the `src\app` folder from the source code of *Chapter 9*, *Navigate through Application with Routing*, in the `src\app` folder of the current Angular project. Replace any files if needed.
3. Copy the `styles.css` file from the source code of *Chapter 9*, *Navigate through Application with Routing*, in the `src` folder of the current Angular project and replace it.
4. Open the `products.module.ts` file and add the following `import` statement:

   ```
   import { FormsModule } from '@angular/forms';
   ```

   We add template-driven forms to an Angular application by importing `FormsModule` from the `@angular/forms` npm package.

5. Add `FormsModule` in the `imports` array of the `@NgModule` decorator:

   ```
   @NgModule({
     declarations: [
       ProductListComponent,
       ProductDetailComponent,
       SortPipe,
       ProductViewComponent,
       ProductCreateComponent
     ],
     imports: [
       CommonModule,
       ProductsRoutingModule,
       FormsModule
     ],
     exports: [ProductListComponent]
   })
   ```

We have already established the infrastructure in our Angular application to start using Angular forms. We will now create our first form for changing the product price in the product details component:

1. Open the `product-detail.component.html` file and modify the `<input>` element as follows:

   ```
   <input placeholder="New price" 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 syntax of the `ngModel` directive is known as *a banana in a box*, which is a memory rule for you to be able to remember how to type it. We create it in two steps. First, we create the banana by surrounding `ngModel` in parentheses `()`. Then, we put the banana in a box by surrounding it with square brackets `[()]`. Remember, it's called banana in a box, not box in a banana.

The `name` attribute is required in the `<input>` element so that Angular can create a unique form control internally to distinguish it.

2. Modify the `<button>` element as follows:

```
<button type="submit">Change</button>
```

In the preceding snippet, we removed the `click` event from the `<button>` element because our form will be responsible for updating the price. We also added the `submit` type to indicate that the form submission can happen by clicking the button.

3. Surround the `<input>` and `<button>` elements with a `<form>` element:

```
<form (ngSubmit)="changePrice(product, product.price)">
  <input placeholder="New price" name="price" [(ngModel)]="product.
price" />
  <button 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 execution of the `changePrice` method 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.

4. Open the `product-detail.component.css` file and add the following CSS style to target the `form` element:

```
form {
  display: inline;
}
```

5. Run the application using the `ng serve` command and select a product from the list.

6. 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 also changes:

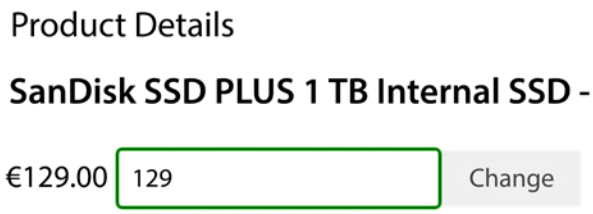


Figure 10.2: Two-way binding

The behavior of our application depicted in the preceding image is the magic behind two-way binding and `ngModel`. 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.

The syntax of a banana in a box that we use for the `ngModel` directive is not random. Under the hood, `ngModel` is a directive that contains an `@Input` binding named `ngModel` and an `@Output` binding named `ngModelChange`. It implements a particular interface called `ControlValueAccessor` that is used to create custom controls for forms. By convention, when a directive or a component contains both bindings that start with the same name, but the output binding ends in `Change`, the property can be used as a two-way binding.

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 details 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. Open the `product-detail.component.html` file and replace the bindings in the `<input>` and `<form>` elements to use the new component property:

```
<form (ngSubmit)="changePrice(product, price!)">
  <input placeholder="New price" name="price" [(ngModel)]="price" />
  <button type="submit">Change</button>
</form>
```

> In the preceding snippet, we use the non-null assertion operator in the form binding because the `price` property has been declared as `number | undefined`.

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 come in handy when creating small and simple forms. In the next section, we dive deeper into the alternative approach offered by the Angular framework: reactive forms.

Using reactive patterns in Angular forms

Reactive forms, as the name implies, provide access to web forms in a reactive manner. 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 state of the form can be modified explicitly and consistently.

Reactive forms have a programmatic approach to creating form elements and setting up validation rules. We set everything up in the component class and merely point out our created artifacts in the template.

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 top-most `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.

All these classes are available from the `@angular/forms` npm package. The `FormControl` and `FormGroup` classes inherit from `AbstractControl`, which contains a lot of interesting properties. We can use these properties to render the UI differently based on what status a particular control or group has. We might want to differentiate UI-wise between a form we have never interacted with and one we have. It could also be interesting to know whether we have interacted with a particular control. As you can imagine, there are many scenarios where it is interesting to know a specific status. We will explore all these properties using the `FormControl` and `FormGroup` classes.

In the next section, we will explore how to work with reactive forms in Angular using our component for creating new products.

Interacting with reactive forms

The Angular application we have built contains a component to add new products for our e-shop. When we built the component in the previous chapter, you may have noticed that the name and price input controls were not cleared upon creating a product. Implementing such functionality would be complex because we would need to access the native `<input>` elements inside the component class. However, Angular forms provide a helpful and convenient API that we can use to accomplish this task. We will learn how to use reactive forms by integrating them into the product create component:

1. Open the `products.module.ts` file and import `ReactiveFormsModule` from the `@angular/forms` npm package:

   ```
   import { FormsModule, ReactiveFormsModule  } from '@angular/forms';
   ```

   The Angular forms library provides `ReactiveFormsModule`, which we can use to start creating reactive forms in our Angular application.

2. Add `ReactiveFormsModule` in the `imports` array of the `@NgModule` decorator:

   ```
   @NgModule({
     declarations: [
       ProductListComponent,
       ProductDetailComponent,
       SortPipe,
       ProductViewComponent,
       ProductCreateComponent
     ],
     imports: [
       CommonModule,
       ProductsRoutingModule,
       FormsModule,
       ReactiveFormsModule
     ],
     exports: [ProductListComponent]
   })
   ```

`ProductsModule` imports both `FormsModule` and `ReactiveFormsModule` in the previous example. There is no harm in doing this. You can use them simultaneously in an Angular application, and everything will work fine.

3. Open the `product-create.component.ts` file and add the following `import` statement:

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

4. Define the following `productForm` property in the `ProductCreateComponent` class:

```
productForm = new FormGroup({
  name: new FormControl('', { nonNullable: true }),
  price: new FormControl<number | undefined>(undefined, {
nonNullable: true })
});
```

5. The constructor of the `FormGroup` class accepts an object that contains key-value pairs of `FormControl` instances. The key denotes a unique name for the form control that `FormGroup` can use to keep track of, while the value is an instance of `FormControl`.
6. The constructor of the `FormControl` class accepts the default value of the input control as the first parameter. For the `name` form control, we pass an empty string so that we do not set any value initially. For the `price` form control that should accept numbers as values, we set it initially to `undefined`.
7. The second parameter passed in the `FormControl` instance is an object that sets the `nonNullable` property to indicate that the form control does not accept null values.
8. 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 component template with the following `<form>` element:

```
<form [formGroup]="productForm">
  <div>
    <label for="name">Name</label>
    <input id="name" #name />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" #price />
  </div>
```

```
  <div>
    <button (click)="createProduct(name.value, price.
valueAsNumber)">Create</button>
  </div>
</form>
```

In the preceding template, we use the `formGroup` directive, exported from `ReactiveFormsModule`, to connect a `FormGroup` instance to a `<form>` element.

9. `ReactiveFormsModule` also exports the `formControlName` directive, which we use to connect a `FormControl` instance to an `<input>` element. Modify the `<input>` elements of the form as follows:

```
<div>
  <label for="name">Name</label>
  <input id="name" formControlName="name" />
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" />
</div>
```

In the preceding snippet, we set the value of the `formControlName` directive to the name of the `FormControl` instance.

Currently, we access the `name` and `price` template variables in the binding of the button `click` event. In reactive forms, this is not the case since the form model is the source of truth. So, we need to get input control values from the `FormGroup` or `FormControl` classes. The `FormGroup` class exposes the `controls` property, which we can use to get a specific `FormControl` instance:

1. Open the `product-create.component.ts` file and create the following getter properties:

```
get name() { return this.productForm.controls.name }
get price() { return this.productForm.controls.price }
```

2. Modify the `createProduct` method so that it uses the newly created properties:

```
createProduct() {
  this.productsService.addProduct(this.name.value, Number(this.
price.value)).subscribe(product => {
    this.productForm.reset();
```

```
      this.added.emit(product);
    });
  }
```

The `FormControl` class contains various properties, such as the value of the associated input control. In the preceding method, we also use the `reset` method of the `productForm` property to reset the form in its initial values.

The `FormGroup` class also contains a `value` property, which we can use to access form control values as a single object. We usually use this property when posting whole entities in a backend API.

3. Open the `product-create.component.html` file and modify its content so that the `createProduct` method is called on form submission:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
  <div>
    <label for="name">Name</label>
    <input id="name" formControlName="name" />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" />
  </div>
  <div>
    <button type="submit">Create</button>
  </div>
</form>
```

If we run the application, we will see that the functionality of adding a new product still works as expected. The **Name** and **Price** fields are also cleared upon creating a new product.

Click the **Create** button without entering any values in the input fields and observe what happens in the **Network** tab inside the developer tools of your browser. The application will try to create a product with an empty name and price set to 0. 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.

In the next section, we'll investigate different properties that we can check to get the status of a form control and provide feedback to the user according to that status.

Providing form status feedback

The Angular framework sets the following CSS classes automatically in a form control according to the current status of the control:

- `ng-untouched`: Indicates that we have not interacted with the control yet
- `ng-touched`: Indicates that we have interacted with the control
- `ng-dirty`: Indicates that we have set a value to the control
- `ng-pristine`: Indicates that the control does not have a value yet
- `ng-valid`: Indicates that the value of the control is valid
- `ng-invalid`: Indicates that the value of the control is not valid

Each class name has a similar property in the form model. The property name is the same as the class name without the `ng-` prefix. We could try to leverage both and provide a unique experience with our forms.

Suppose we would like to display a highlighted border in an input control when interacting with that control for the first time. We should define a global CSS style in the `styles.css` file, such as the following:

```
input.ng-touched {
  border: 3px solid lightblue;
}
```

We can also combine some of the CSS classes according to the needs of our application. Suppose we would like to display a green border when an input control has a value and a red one when it does not have any at all. The red border should be visible only if we initially entered a value in the input control and deleted it immediately afterward. We should create the following CSS rules in the `styles.css` file:

```
input.ng-dirty.ng-valid {
  border: 2px solid green;
}

input.ng-dirty.ng-invalid {
  border: 2px solid red;
}
```

We must add a validation rule to our input elements for the preceding classes to work. We can use many built-in Angular validators, as we will learn later in this chapter. In this case, we will use the `required` validator, which indicates that an input control must have a value to be valid. To apply it, add the `required` attribute to both `<input>` elements in the `product-create.component.html` file:

```
<form [formGroup]="productForm" (ngSubmit)="createProduct()">
  <div>
    <label for="name">Name</label>
    <input id="name" formControlName="name" required />
  </div>
  <div>
    <label for="price">Price</label>
    <input id="price" formControlName="price" required />
  </div>
  <div>
    <button type="submit">Create</button>
  </div>
</form>
```

Later, in the *Validating controls in a reactive way* section, we will learn how to apply a validator to a `FormControl` instance directly.

Run the application using the `ng serve` command and follow these steps to check the applied CSS rules:

1. Click on the **Name** field and then on the **Price** field. The former should now display a light blue border.
2. Enter some text into the **Name** field and click outside the input control. Notice that it has a green border.
3. Remove the text from the **Name** field and click outside the input control. The border should now turn red.
4. Repeat all previous steps for the **Price** field.

We can now understand what happens when the status of an input control changes and notify users visually about that change. In the next section, we'll learn that the status of a form can be spawned across many form controls and groups at different levels.

Creating nesting form hierarchies

We have already seen how to create a form with two input controls. The product create form is a simple form that needs one `FormGroup` and two `FormControls`. There are use cases in enterprise applications that require more advanced forms that involve creating nested hierarchies of form groups. Consider the following form, which is used to add product information along with basic details:

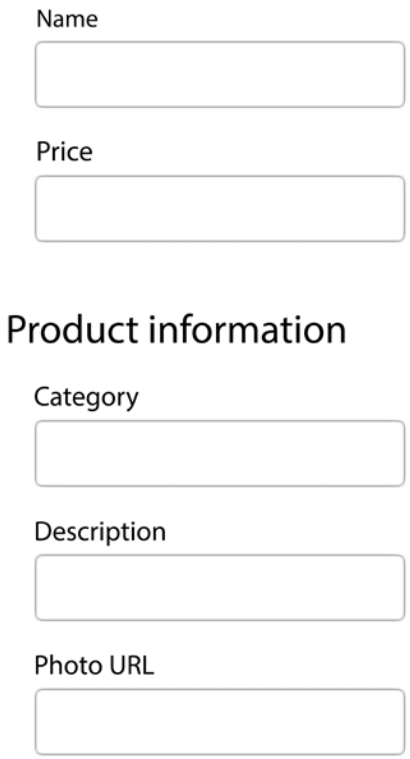


Figure 10.3: New product form

The preceding form may look like a single one, but if we take a better look at the component class, we will see that it consists of two `FormGroup` instances, one nested inside the other:

```
productForm = new FormGroup({
  name: new FormControl('', {
    nonNullable: true
  }),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true
```

```
    }),
    info: new FormGroup({
      category: new FormControl(''),
      description: new FormControl(''),
      image: new FormControl('')
    })
  });
```

The `productForm` property is the parent form group, while `info` is its child. A parent form group can have as many children form groups as it wants. 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 formGroupName="info">
  <h2>Product information</h2>
  <div>
    <label for="category">Category</label>
    <input id="category" formControlName="category" />
  </div>
  <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>
```

In the preceding HTML template, we use the `formGroupName` directive to bind the inner form element to the `info` property.

You may have expected to bind it directly to the `productForm.info` property, but this will not work. The Angular framework is pretty smart because it understands that `info` is a child form group of `productForm`. It can deduce this information because the form element related to `info` is inside the form element that binds to the `productForm` property.

The status of a child form is shared with its parent in a nested form hierarchy. In our case, when the `info` form becomes invalid, its parent form, `productForm`, will also be invalid. The change of status is not the only thing that bubbles up to the parent form. The value of the child form also propagates up the hierarchy, thereby maintaining a consistent form model:

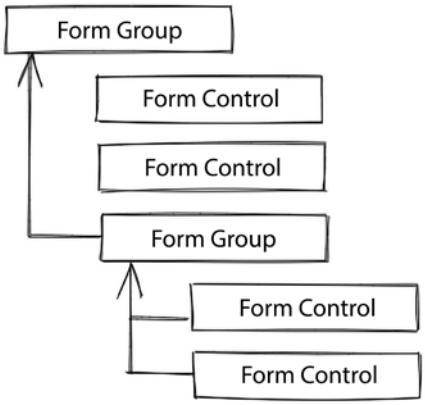


Figure 10.4: Status and value propagation in nested forms hierarchy

Nested hierarchies add a useful feature for Angular forms to the developer's toolchain when organizing large form structures. The status and value of each form propagate through the hierarchy to provide stability to our models.

So far, we have been using the constructor of `FormGroup` and `FormControl` classes to create an Angular form. However, it constitutes a lot of noise, especially in forms that contain many controls. In the following section, we will learn how to create Angular forms using an Angular service.

Creating elegant reactive forms

The Angular forms library exposes a service called `FormBuilder` that we can use to simplify form creation. We will learn how to use it by converting the form we created in the product create component:

1. Open the `product-create.component.ts` file and import the `FormBuilder` artifact from the `@angular/forms` npm package:

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

2. Inject `FormBuilder` in the `constructor` of the `ProductCreateComponent` class:

```
constructor(private productsService: ProductsService, private
builder: FormBuilder) {}
```

3. Convert the `productForm` property as follows:

```
productForm: FormGroup<{
  name: FormControl<string>,
  price: FormControl<number | undefined>
}> | undefined;
```

4. Create the following `buildForm` method:

```
private buildForm() {
  this.productForm = this.builder.nonNullable.group({
    name: this.builder.nonNullable.control(''),
    price: this.builder.nonNullable.control<number |
undefined>(undefined, {})
  });
}
```

   We use the `group` method of the `FormBuilder` service to group form controls together. We also use its `control` method to create the form controls. Notice that we also use the `nonNullable` property to indicate that the form and its controls are not nullable.

5. Make sure you use the non-null assertion operator in all references of the `productForm` property because it does not have an initial value anymore.

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.

We have already seen how to define validation rules in a template by triggering a change of status using CSS styling. In the next section, we will learn how to define them in the component class and give visual feedback using appropriate messages.

Validating controls in a reactive way

We have already learned how to apply validation to the template of a form. We used the `required` attribute in the *Using reactive patterns in Angular forms* section to indicate that an input control needs to have a value. In reactive forms, the source of truth is our form model, so we need to be able to define validation rules when building the `FormGroup` instance.

To add validation rules, we use the second parameter of the `FormControl` constructor:

1. Open the `product-create.component.ts` file and import the `Validators` artifact from the `@angular/forms` npm package:

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

2. Modify the declaration of the `productForm` property so that each `FormControl` instance passes `Validators.required` as a second parameter:

```
productForm = new FormGroup({
  name: new FormControl('', {
    nonNullable: true,
    validators: Validators.required
  }),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true,
    validators: Validators.required
  })
});
```

When we add a validator using the constructor of `FormControl`, we can remove the respective HTML attribute from the HTML template. However, it is recommended to keep it for accessibility purposes so that screen readers can understand how the form control should be validated.

3. The `Validators` class contains almost the same validator rules that are available for template-driven forms, such as the required validator. We can combine multiple validators by adding them to an array. To configure the price form control so that its value is at least 1, we use the `Validators.min` method:

```
productForm = new FormGroup({
  name: new FormControl('', {
    nonNullable: true,
    validators: Validators.required
  }),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true,
    validators: [Validators.required, Validators.min(1)]
```

```
  })
});
```

4. We can now use the status of validation rules and react to their changes. To disable the `<button>` element when the form is not valid, we need to bind the status of the form to the `disabled` button property in the `product-create.component.html` file:

```
<div>
  <button type="submit" [disabled]="!productForm.valid">Create</
button>
</div>
```

5. We can also display specific messages to the user upon changing the validity of each control:

```
<div>
  <label for="name">Name</label>
  <input id="name" formControlName="name" required />
  <span *ngIf="name.touched && name.invalid">
    The name is not valid
  </span>
</div>
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" required />
  <span *ngIf="price.touched && price.invalid">
    The price is not valid
  </span>
</div>
```

6. It would be nice, though, if we could display different messages depending on the validation rule. We could display a more specific message when the price is less than 1. The `FormControl` class contains the `hasError` method, which accepts the validation property as a parameter and checks if the control has set the particular validation error:

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" required />
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
```

```
  </span>
  <span *ngIf="price.touched && price.hasError('min')">
    The price should be greater than 1
  </span>
</div>
```

The Angular framework provides a set of built-in validators that we can use in our forms. A validator is a function that returns either a `ValidationErrors` object or `null` when the control does not have any errors. According to the scenario, a validator can also return a value synchronously or asynchronously. In the following section, we will learn how to create a custom synchronous validator.

Building a custom validator

Sometimes, default validators won't cover all the scenarios we might encounter in an Angular application. It is easy to write a custom validator and use it in an Angular reactive form. In our case, we will build a validator to check whether the price of a product is in a predefined amount range.

We have already learned that a validator is a function that needs to return a `ValidationErrors` object with the error specified or a `null` value. Let's define such a function in a file named `price-range.directive.ts` inside the `src\app\products` folder:

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/
forms';

export function priceRangeValidator(): ValidatorFn {
  return (control: AbstractControl<number>): ValidationErrors | null => {
    const inRange = control.value > 1 && control.value < 10000;
    return inRange ? null : { outOfRange: true };
  };
}
```

The validator is a function that returns another function called the configured validator function. It accepts the form control object to which it will be applied as a parameter. If the value of the control does not fall into a defined price range, 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.

To use our new validator, all we must do is import it into our product create component and add it to the `price` `FormControl` instance:

```
productForm = new FormGroup({
  name: new FormControl('', {
    nonNullable: true,
    validators: Validators.required
  }),
  price: new FormControl<number | undefined>(undefined, {
    nonNullable: true,
    validators: [Validators.required, priceRangeValidator()]
  })
});
```

We removed the min validator from the price control because it is already checked from the functionality of our price range validator.

We can now modify the `price` field in the `product-create.component.html` file to display an appropriate error message if that specific error occurs:

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" required />
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
  </span>
  <span *ngIf="price.touched && price.hasError('outOfRange')">
    The price is out of range
  </span>
</div>
```

Angular forms are not only about checking statuses but also about setting values. In the next section, we'll learn how to programmatically set values in a form.

Modifying forms dynamically

So far, we have used the `FormGroup` and `FormControl` classes extensively throughout this chapter. However, we have not seen what `FormArray` is all about.

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. Create a new service to manage the shopping cart by running the following Angular CLI command in the `src\app\cart` folder:

```
ng generate service cart
```

2. Open the `cart.service.ts` file and add the following `import` statement:

```
import { Product } from '../products/product';
```

3. Create a `cart` property in the `CartService` class and initialize it to an empty array:

```
export class CartService {

  cart: Product[] = [];

  constructor() { }
}
```

The preceding `cart` property will be an intermediate local cache for storing selected products before checking out.

4. Add the following method to add a product to the cart:

```
addProduct(product: Product) {
  this.cart.push(product);
}
```

5. Open the `product-detail.component.ts` file and add the following `import` statement:

```
import { CartService } from '../../cart/cart.service';
```

6. Inject `CartService` in the `ProductDetailComponent` class:

```
constructor(
  private productService: ProductsService,
  public authService: AuthService,
  private route: ActivatedRoute,
  private cartService: CartService
) { }
```

7. Modify the `buy` method to call the `addProduct` method of the `CartService` class:

```
buy(product: Product) {
  this.cartService.addProduct(product);
}
```

8. Finally, open the `product-detail.component.html` file and modify the `Buy Now` button:

```
<button *ngIf="authService.isLoggedIn" (click)="buy(product)">Buy
Now</button>
```

Now that we have implemented the basic functionality for storing the selected products that users want to buy, we need to modify the cart component for displaying the cart items:

1. Open the `cart.component.ts` file and add the following `import` statements:

```
import { FormArray, FormControl, FormGroup } from '@angular/forms';
import { Product } from '../products/product';
import { CartService } from './cart.service';
```

2. Create the following properties in the `CartComponent` class:

```
cartForm = new FormGroup({
  products: new FormArray<FormControl<number>>([])
});
cart: 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 type `number` as a parameter. For now, the list is empty since the cart does not have any products initially.

3. We have also created a `cart` property to store the details of the current shopping cart.
4. Inject `CartService` in the `constructor` of the `CartComponent` class:

```
constructor(private cartService: CartService) { }
```

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 `CartComponent` class:

```
export class CartComponent implements OnInit {
```

7. Add the following `ngOnInit` method:

```
ngOnInit(): void {
  this.cart = this.cartService.cart;
  this.cart.forEach(() => {
    this.cartForm.controls.products.push(
      new FormControl(1, { nonNullable: true })
    );
  });
}
```

In the preceding method, we get the `cart` property from the `CartService` class and store it in the `cart` component property. We iterate through the products of the shopping cart and add the respective `FormControl` instances in the `FormArray`. We set the value of each form control to `1` to indicate that the cart contains one piece from each product by default.

8. Open the `cart.component.html` file and replace its HTML template with the following content:

```
<h2>My Cart</h2>
<div [formGroup]="cartForm">
  <div
    formArrayName="products"
    *ngFor="let product of cartForm.controls.products.controls; let
i=index">
      <label>{{cart[i].name}}</label>
      <input type="number" [formControlName]="i" />
  </div>
</div>
```

In the preceding template, we use the `*ngFor` directive 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 `*ngFor` directive to give a dynamically created name to each form control using the `formControlName` binding. We have also added a label that displays the product name from the `cart` component property. The product name is fetched using the index of the current product in the array.

9. Open the `app.module.ts` file and add the following `import` statements:

```
import { ReactiveFormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';
```

10. Add `ReactiveFormsModule` and `CommonModule` in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    AppComponent,
    CartComponent,
    PageNotFoundComponent
  ],
  imports: [
    BrowserModule,
    ProductsModule,
    AppRoutingModule,
    HttpClientModule,
    AuthModule,
    ReactiveFormsModule,
    CommonModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

To see the cart component in action, run the application using the `ng serve` command. Add some products to the cart by navigating to their details page and clicking the **Buy Now** button.

Do not forget to log in using the **Login** button because the functionality that adds a product to the cart is available only to authenticated users.

After you have added some products to the cart, click the **Cart** link to view your shopping cart. It should look like the following:

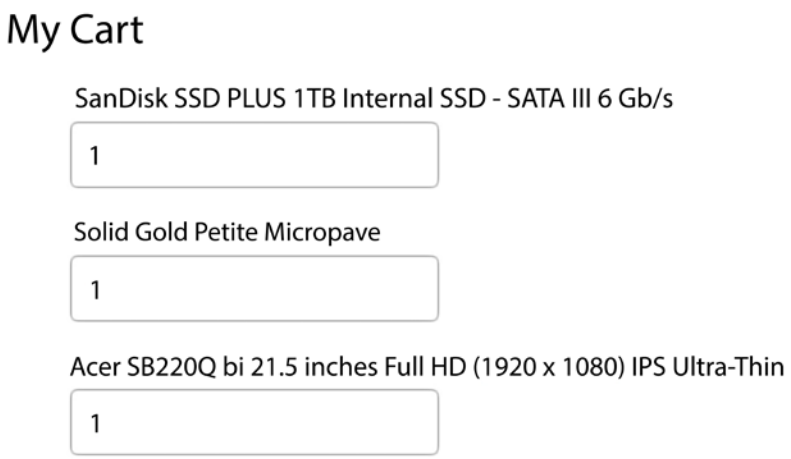


Figure 10.5: Shopping cart

The real power of the `FormArray` class is that it can be used not only with `FormControl` instances but also with more complicated structures and other form groups.

With the `FormArray`, we have completed our knowledge range about the most basic building blocks of an Angular form. In the next section, we'll learn how to use the reactive forms API and set values programmatically to an Angular form.

Manipulating form data

The `FormGroup` class contains two methods that we can use to change the values of a form programmatically:

- `setValue`: Replaces values in all the 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, we should use the following snippet:

```
this.productForm.setValue({
  name: 'New product',
  price: 150
});
```

In the preceding snippet, each key of the object passed in the `setValue` method must match the name of each control in the form. If we omit one, Angular will throw an error.

If, on the contrary, we want to fill in some of the details of a product, we can use the `patchValue` method:

```
this.productForm.patchValue({
  price: 150
});
```

The `setValue` and `patchValue` methods of the `FormGroup` class help us set data in a form. Another interesting aspect of reactive forms is that we can also be notified when these values change, as we will see in the following section.

Watching state changes and being reactive

We have already learned how to create forms programmatically and specify all our fields and their validations in the code. A reactive form can listen to changes in the form controls when they happen and react accordingly. A suitable reaction could be to disable/enable a control, provide a visual hint, or do something else according to your needs.

How can we make this happen? A `FormControl` instance contains two observable properties: `statusChanges` and `valueChanges`. The first one notifies us when the status of the control changes, such as going from invalid to valid. On the other hand, the second one notifies us when the value of the control changes. Let's explore this one in more detail, using an example.

The **Price** field in the form of the product create component contains a custom validator to check if the price is within a valid range. From an end-user point of view, it would be better to display a hint about this validation as soon as the user has started entering values in the field:

1. First, add a `<span>` element in the `product-create.component.html` file to contain an appropriate hint message:

```
<div>
  <label for="price">Price</label>
  <input id="price" formControlName="price" required />
  <span *ngIf="price.touched && price.hasError('required')">
    The price is required
  </span>
  <span *ngIf="price.touched && price.hasError('outOfRange')">
    The price is out of range
```

```
    </span>
    <span *ngIf="showPriceRangeHint">
      Price should be between 1 and 10000
    </span>
  </div>
```

The hint will be displayed according to the `showPriceRangeHint` property of the component.

2. Open the `product-create.component.ts` file and import the `OnInit` artifact:

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

3. Add the `OnInit` artifact in the list of the `ProductCreateComponent` class implemented interfaces:

```
export class ProductCreateComponent implements OnInit {
```

4. Create the `showPriceRangeHint` property in the `ProductCreateComponent` class:

```
showPriceRangeHint = false;
```

5. Create the following `ngOnInit` method to subscribe to the `valueChanges` property of the `price` form control:

```
ngOnInit(): void {
  this.price.valueChanges.subscribe(price => {
    if (price) {
      this.showPriceRangeHint = price > 1 && price < 10000;
    }
  });
}
```

In the preceding method, we check if the price entered is within a valid range and set the `showPriceRangeHint` property appropriately.

The `valueChanges` and `statusChanges` properties in a `FormControl` instance are standard observable streams. Do not forget to unsubscribe from them 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 name is already reserved by sending it to a backend server, but this code shows off the reactive nature. Hopefully, this has conveyed how you can take advantage of the reactive nature of forms and respond accordingly.

Summary

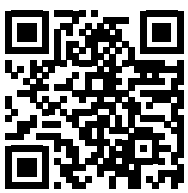
In this chapter, we have learned that Angular provides two different flavors for creating forms – template-driven and reactive forms – and that neither approach can be said to be better than the other. We have merely focused on reactive forms because of their many advantages and learned how to build them. We have also covered the different types of validations and now know how to create our custom validations. We also learned how to fill our forms with values and get notified when they change.

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 styling ready for you to use in your projects. So, let's give your Angular project the love it deserves.

Join our community on Discord

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

`https://packt.link/LearningAngular4e`



11

Introduction to Angular Material

When developing a web application, you must decide how to create your UI. It should 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. Undoubtedly, most developers consider creating the UI/UX a daunting task and, therefore, turn to UI frameworks that do much of the heavy lifting. Some frameworks are used more than others, namely the following:

- **Bootstrap**
- **Tailwind CSS**

However, there is a new kid on the block—**Angular Material**—based on Google's **Material Design** techniques. In this chapter, we will explain what Material Design is and how Angular Material implements its principles, and we will also look at some of its core components. In this chapter, we will be doing the following:

- Introducing Material Design
- Introducing Angular Material
- Adding core UI controls
- Introducing the Angular CDK

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 `ch11` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Introducing Material Design

Material Design is a design language that Google developed in 2014. Google states that its new design language is based on paper and ink. The creators of Material Design explained their goal in the following way:

> *We challenged ourselves to create a visual language for our users that synthesizes the classic principles of good design with the innovation and possibility of technology and science.*

They further explained their goals as follows:

- Develop a single underlying system that allows for 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.

All in all, it can be said that there is much theory behind the design language, and there is proper documentation on the topic should you wish to delve further. You can find more information at the official documentation site, `https://material.io`.

Now, this is probably very interesting if you are a designer. But we are web developers—why should we bother looking at this at all? Every time Google sets out to build something, it becomes big, and while not everything stays big forever, there is sufficient muscle behind it to indicate that Material Design will be around for quite a while. Google has extensively used it in their products, such as Firebase, Gmail, and Google Analytics.

But of course, a design language by itself isn't that interesting, at least not for a developer, which brings us to the following section, where we will learn about the most known implementation based on Material Design principles, the Angular Material library.

Introducing Angular Material

The Angular Material library was developed to implement Material Design for the Angular framework. It promotes itself with the following features:

- **Sprint from zero to app**: The intention is to make it easy for you as an application developer to hit the ground running. The amount of effort in setting it up should be minimal.
- **Fast and consistent**: Performance has been a significant focus point, and it is guaranteed to work well on all major browsers.
- **Versatile**: Many themes should be easy to customize, 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 parts:

- **Components**: There are a ton of UI components in place to help you be successful, such as different kinds of input, buttons, layout, navigation, modals, and different ways to show tabular data.
- **Themes**: The library comes with a set of preinstalled themes, but there is also a theming guide if you want to create your own at `https://material.angular.io/guide/theming`.
- **Icons**: Material Design comes with over 900 icons, so you will likely find exactly the icon you need. You can browse through the full collection at `https://fonts.google.com/icons?selected=Material+Icons`.

We have already 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.

Adding Angular Material to your application

The Angular Material library is an npm package. To install it, we need to manually execute the `npm install` command and import several Angular modules into our Angular application. Luckily, the Angular team has automated these interactions by creating the necessary schematics to install it using the Angular CLI.

We can use the `ng add` Angular CLI command to install Angular Material in an existing Angular application:

```
ng add @angular/material
```

Initially, 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 15 which is compatible with Angular 15. If the version that prompts you is different, you should run the command ng add `@angular/material@15` to install the latest Angular Material 15 to your system.

After the download completes successfully, it will ask us whether we want to use a prebuilt theme for our Angular application or a custom one. Accept the default value **Indigo/Pink** and press *Enter*:

```
? Choose a prebuilt theme name, or "custom" for a custom theme: (Use arrow keys)
> Indigo/Pink        [ Preview: https://material.angular.io?theme=indigo-pink ]
  Deep Purple/Amber  [ Preview: https://material.angular.io?theme=deeppurple-amber ]
  Pink/Blue Grey     [ Preview: https://material.angular.io?theme=pink-bluegrey ]
  Purple/Green       [ Preview: https://material.angular.io?theme=purple-green ]
  Custom
```

Figure 11.1 : Theme selection

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. Angular Material typography is based on Material Design guidelines and uses the **Roboto** Google font for styling.

We want to keep our application as simple as possible, so accept the default value, **No**, by pressing *Enter*:

```
? Set up global Angular Material typography styles? (y/N)
```

Figure 11.2 : Set up typography

The next question is about animations. We want our application to display a beautiful animation when we click on a button or open a modal dialog. It isn't strictly required, but we want some cool animations. Accept the default value, **Include and enable animations**, by pressing *Enter*:

```
? Include the Angular animations module? (Use arrow keys)
> Include and enable animations
  Include, but disable animations
  Do not include
```

Figure 11.3: Set up animations

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. After the process is finished, we can begin adding controls from the Angular Material library into our application.

Adding Angular Material controls

To start using a UI control from the Angular Material library, such as a button or a checkbox, we need to import its corresponding module. Let's see how this is done by adding a button control in the main component of an Angular application:

1. Open the `app.module.ts` file and add the following `import` statement to use Angular Material buttons:

   ```
   import { MatButtonModule } from '@angular/material/button';
   ```

   We do not import directly from `@angular/material` because every module has a dedicated namespace. The button controls can be found in the `@angular/material/button` namespace.

2. Add `MatButtonModule` in the `imports` array of the `@NgModule` decorator:

   ```
   @NgModule({
     declarations: [
       AppComponent
     ],
     imports: [
       BrowserModule,
       BrowserAnimationsModule,
       MatButtonModule
     ],
     providers: [],
     bootstrap: [AppComponent]
   })
   ```

3. Open the `app.component.html` file and replace its content with the following HTML template:

   ```
   <button mat-button>I am an Angular Material button</button>
   ```

In the preceding template, we added a `<button>` element and attached the `mat-button` directive to it. The `mat-button` directive, in essence, modifies the `<button>` element so that it resembles and behaves as a Material Design button.

That's it! We now have a simple Angular application that is decorated with Material Design. But there is more—much more. For instance, we can apply several colors to the button we created according to the selected theme, which is the topic of the following section.

Theming Angular Material components

As we saw in the previous section, the Angular Material library comes with four built-in themes:

- Indigo/Pink
- Deep Purple/Amber
- Pink/Blue-Gray
- Purple/Green

When we add Angular Material to an Angular CLI project, the **Indigo/Pink** theme is the default one. We can always change the selected theme by modifying the included CSS stylesheet in the `angular.json` configuration file:

```
"styles": [
  "/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.css"
]
```

Each theme consists of a set of color palettes, the most common ones being the following:

- Primary
- Accent
- Warn

So, if we want to apply the **primary** palette to our button, we would modify the HTML template as follows:

```
<button mat-button color="primary">
  I am an Angular Material button
</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 magical journey through the land of styling with Angular Material, we will discuss some of the essential core components in the next section.

Adding core UI controls

Angular Material consists of many components of different types. Some of the most basic ones are:

- **Buttons**: These are what they sound like – buttons you can push. But there are several different types that you can use, such as icon buttons, raised buttons, and more.
- **Form controls**: These are any control that we use to collect data from a form, such as autocomplete, checkbox, input, radio button, and drop-down list.
- **Navigation**: These are controls used to perform navigation, such as a menu, a sidenav, or a toolbar.
- **Layout**: These are controls that define how data is arranged on a page, such as a list, a card, or tabs.
- **Popups/modals**: These are overlay windows that block any user interaction until they are dismissed in any way.
- **Tables**: These are controls that are used to display data in a tabular way. What kind of table you need depends on whether your data is massive, needs pagination, needs to be sorted, or all of these.
- **Integration controls**: These are controls that integrate external services in an Angular Material application like Google Maps and YouTube.

In the following sections, we will explore each category in more detail.

Buttons

We have already learned how to create a simple button with Angular Material. There are, however, a lot more button types, namely the following:

- `mat-raised-button`: A button displayed with a shadow to indicate its raised state. A variation of this button is `mat-flat-button`, which is the same button but without a shadow.
- `mat-stroked-button`: A button with a border.
- `mat-icon-button`: A button that displays an icon only, without text.
- `mat-fab`: A rounded button with an icon. A variation of this type is `mat-mini-fab`, which displays a smaller button.
- `mat-button-toggle`: A button with on/off capabilities that indicates whether it has been pressed or not.

In the following snippet, you can see how to use each button type:

```
<button mat-raised-button>Raised button</button>
<button mat-flat-button>Flat button</button>
<button mat-stroked-button>Stroked button</button>
<button mat-icon-button>
  <mat-icon>favorite</mat-icon>
</button>
<button mat-fab>
  <mat-icon>delete</mat-icon>
</button>
<mat-button-toggle-group>
  <mat-button-toggle value="left">
    <mat-icon>format_align_left</mat-icon>
  </mat-button-toggle>
  <mat-button-toggle value="center">
    <mat-icon>format_align_center</mat-icon>
  </mat-button-toggle>
  <mat-button-toggle value="right">
    <mat-icon>format_align_right</mat-icon>
  </mat-button-toggle>
</mat-button-toggle-group>
```

There are some things to note in setting up the buttons in the preceding snippet:

- To use a `mat-icon` button, we must import `MatIconModule` from the `@angular/material/icon` namespace and add a `<mat-icon>` element inside the button. The content of a `<mat-icon>` element is text that indicates which icon to display. Each icon on the Material Design website contains an image and a piece of descriptive text. We need to insert the appropriate text inside the `<mat-icon>` element to use a specific image.
- A `<mat-icon>` element is also the basis for the `mat-fab` and `mat-mini-fab` buttons, but it can also be used with other button types. Use your imagination to create great buttons.
- To use a `<mat-button-toggle>` element, we need to import `MatButtonToggleModule` from the `@angular/material/button-toggle` namespace. A `<mat-button-toggle>` element is rarely used standalone; instead, it is combined with other buttons of the same type in a `<mat-button-toggle-group>` element.

The resulting output is shown in the following image:

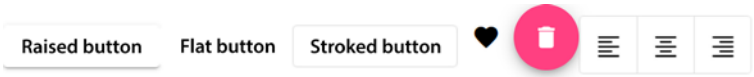


Figure 11.4: Angular Material button types

Buttons are a fundamental element of the Angular Material library. In the following section, we will learn about some Angular Material controls suitable for forms.

Form controls

As we learned in *Chapter 10*, *Collecting User Data with Forms*, 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 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 a list of suggestions while typing. It helps to narrow down the possible values that the input can take.
- **Checkbox**: A classic checkbox that represents 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 either on or off.

In the following sections, we will look at some of the previous form controls in more detail using the source code of *Chapter 10*, *Collecting User Data with Forms*.

In this chapter, we will not use the global CSS styles from the `styles.css` file as we did in previous chapters because we will use Angular Material styling. Additionally, we will apply Angular Material components to selected parts in our e-shop application. However, we encourage you to convert the whole application by yourself.

Input

The input field is a classic input control on which we can set different validation rules. We can easily add the ability to display errors in the input field nicely and reactively. To learn how to use the input control, we will use it in the form of our product's create component:

1. Open the `products.module.ts` file and add the following `import` statements:

```
import { MatButtonModule } from '@angular/material/button';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
```

2. Add the `MatButtonModule`, `MatFormFieldModule`, and `MatInputModule` classes in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent,
    SortPipe,
    ProductViewComponent,
    ProductCreateComponent
  ],
  imports: [
    CommonModule,
    ProductsRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    MatButtonModule,
    MatFormFieldModule,
    MatInputModule
  ],
  exports: [ProductListComponent]
})
```

3. Open the `product-create.component.html` file and modify the `<input>` elements as follows:

```
<div>
  <mat-form-field>
    <input formControlName="name" placeholder="Name" required
matInput />
    <mat-error>The name is not valid</mat-error>
  </mat-form-field>
</div>
<div>
  <mat-form-field>
    <input formControlName="price" placeholder="Price" required
matInput />
    <mat-error>The price is required</mat-error>
    <mat-hint>
      Price should be between 1 and 10000
    </mat-hint>
  </mat-form-field>
</div>
```

   In the preceding HTML snippet, we use the `matInput` directive to indicate that `<input>` elements are Angular Material input controls. We surround them in a `<mat-form-field>` element and add `<mat-error>` and `<mat-hint>` elements to display error and hint messages respectively. A `<mat-error>` element is displayed by default when a validation rule is violated. A `<mat-hint>` element is always displayed in input controls.

4. Add the `mat-raised-button` directive to the `<button>` element so that it is styled as an Angular Material button:

```
<div>
  <button mat-raised-button color="primary" type="submit"
[disabled]="!productForm.valid">Create</button>
</div>
```

If we run the application using the `ng serve` command, the product creation form should look like the following:

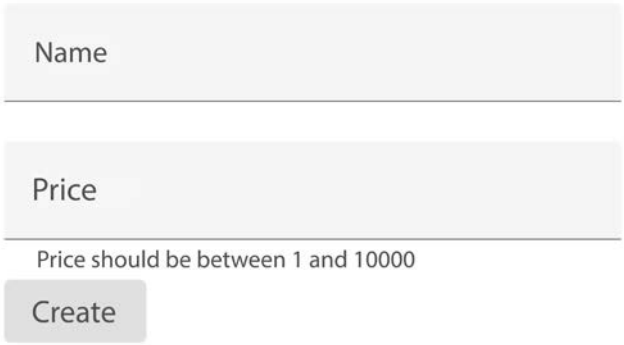


Figure 11.5: Product creation form

In the following section, we will learn how to combine an input control with an autocomplete control to suggest values to the user.

Autocomplete

The idea with autocomplete is to help the user narrow down the possible values that an input field can have. In a regular input field, you would type something and hope a validation tells you whether what you have entered is correct. With autocomplete, you are presented with a list of values you are most likely to want as you type, and at any point, you can decide to stop typing and select an item from the list. It is a time saver, as you don't have to type the entire item's name, and it also enhances accuracy because typing is often error-prone.

To learn how autocomplete works, we will provide a list of existing product names in the name input control so that users do not add a product that already exists:

1. Open the `products.module.ts` file and add the following `import` statement:

```
import { MatAutocompleteModule } from '@angular/material/
autocomplete';
```

2. Add `MatAutocompleteModule` in the `imports` array of the `@NgModule` decorator:

```
@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent,
    SortPipe,
```

```
    ProductViewComponent,
    ProductCreateComponent
  ],
  imports: [
    CommonModule,
    ProductsRoutingModule,
    FormsModule,
    ReactiveFormsModule,
    MatButtonModule,
    MatFormFieldModule,
    MatInputModule,
    MatAutocompleteModule
  ],
  exports: [ProductListComponent]
})
```

3. Open the `product-create.component.ts` file and add the following property in the `ProductCreateComponent` class:

```
products: Product[] = [];
```

4. Assign the `getProducts` method of the `ProductsService` class to the newly created property inside the `ngOnInit` method:

```
this.productsService.getProducts().subscribe(products => {
  this.products = products;
});
```

5. Open the `product-create.component.html` file and add the following HTML snippet right after the first `<mat-form-field>` element:

```
<mat-autocomplete #productsAuto="matAutocomplete">
  <mat-option *ngFor="let product of products" [value]="product.
name">
    {{product.name}}
  </mat-option>
</mat-autocomplete>
```

The `<mat-autocomplete>` element contains a set of `<mat-option>` elements that represent the list of suggested values. We use the `NgFor` directive to iterate over the list of products.

We also define a `productsAuto` template reference variable and bind it to `matAutocomplete` so we can reference it later in the input field. This way, when the input control is focused, it will trigger the autocomplete control to display the suggested product names.

6. Modify the `<input>` element for the product name to connect it with the autocomplete control:

```
<input formControlName="name" placeholder="Name"
[matAutocomplete]="productsAuto" required matInput />
```

If we run our application with the `ng serve` command and focus on the **Name** input, a drop-down list will appear that contains the suggested values from the autocomplete control:

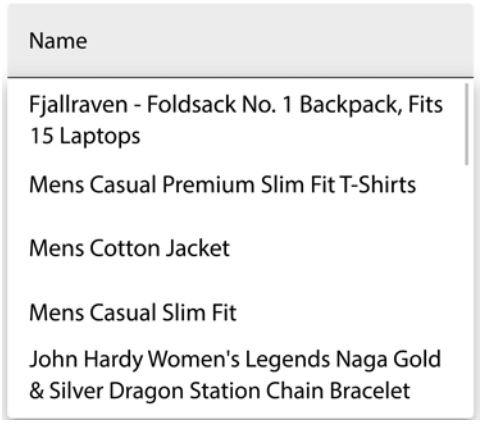


Figure 11.6: Autocomplete control

We are halfway there. Currently, the autocomplete control displays all suggested values. Ideally, we would like to filter them as we type. Specifically, we want to display products whose names start with the text we type in the input control. To accomplish that task, we can use the `valueChanges` observable and subscribe to get notified when the user types in the input control. As soon as the observable emits a new value, we can filter the product list according to that value:

1. Add the following `import` statement in the `product-create.component.ts` file:

```
import { map, Observable } from 'rxjs';
```

2. Create the following `Observable` property in the `ProductCreateComponent` class:

```
products$: Observable<Product[]> | undefined;
```

3. Assign the `products$` property to the `valueChanges` observable in the `ngOnInit` method as follows:

```
this.products$ = this.name.valueChanges.pipe(
  map(name => this.products.filter(product => product.name.
startsWith(name)))
);
```

4. Now we need to change our template so that the `<mat-option>` element iterates over the `products$` observable using the `async` pipe:

```
<mat-autocomplete #productsAuto="matAutocomplete">
  <mat-option *ngFor="let product of products$ | async"
[value]="product.name">
    {{product.name}}
  </mat-option>
</mat-autocomplete>
```

If we run the application and start typing the character `Me` in the input control, we can see that it displays all products whose name starts with **Me** as suggested values:

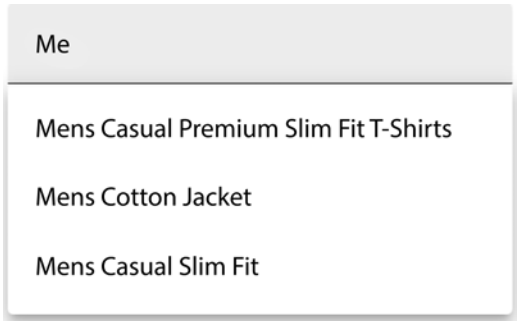


Figure 11.7: Autocomplete filtering

We could have implemented a more advanced filtering mechanism, such as a case-insensitive search, through all the names or a live search. Imagine that, instead of filtering a local array, we sent a request to a backend API and got live results. The possibilities are endless. The only limit is your imagination in crafting good user experiences with autocomplete control.

In the following section, we will learn how to add a `select` component in the form for selecting multiple product categories.

Select

The `select` component works similarly to the native `<select>` element defined in the HTML standard. It displays a drop-down element with a list of options that users can choose. It allows selecting only one option at a time by default, but it can also be configured for multiple selections, as we will see in this example.

We will add a `select` component in the `ProductCreate` component to add multiple categories to a new product:

1. Open the `products.module.ts` file and add the following `import` statement:

```
import { MatSelectModule } from '@angular/material/select';
```

2. Add the `MatSelectModule` class in the `imports` array of the `@NgModule` decorator:

```
imports: [
  CommonModule,
  ProductsRoutingModule,
  FormsModule,
  ReactiveFormsModule,
  MatButtonModule,
  MatFormFieldModule,
  MatInputModule,
  MatAutocompleteModule,
  MatSelectModule
]
```

3. Open the `product-create.component.ts` file and add a `categories` property in the `ProductCreateComponent` class:

```
categories = ['Hardware', 'Computers', 'Clothing', 'Software'];
```

4. To display the category list in the component template, open the `product-create.component.html` file and add the following snippet after the `<div>` element of the `price` field:

```
<div>
  <mat-form-field>
    <mat-label>Categories</mat-label>
    <mat-select multiple>
```

```
      <mat-option *ngFor="let category of categories"
[value]="category">
        {{category}}
      </mat-option>
    </mat-select>
  </mat-form-field>
</div>
```

In the preceding snippet, we use a `<mat-select>` element with the `multiple` directive attached to it. We iterate over the `categories` property and create a `<mat-option>` element for each category item.

If we run the application using the `ng serve` command, we will see that our form has a new control called **Categories**. If we click on it, it displays a list of product categories that we can select:

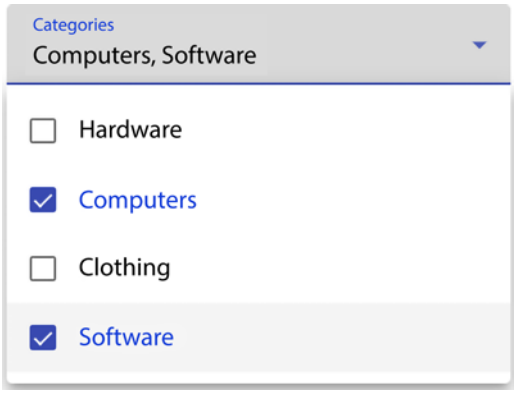


Figure 11.8: Multi-select control

In the following section, we will explore using a checkbox control from the Angular Material library.

Checkbox

The checkbox is a *tristate* control and can have checked, unchecked, or undetermined values. To use it, we first need to import `MatCheckboxModule` from the `@angular/material/checkbox` namespace and then add a `<mat-checkbox>` element to a component template:

```
<mat-checkbox color="primary" [checked]="isChecked">Check me</mat-
checkbox>
```

In the previous snippet, we added a property binding to the `checked` property of the checkbox control to indicate whether it is checked using the `isChecked` component property.

We will finally complete our walkthrough using the form controls of the Angular Material library by looking at the date-picker control in the following section.

Date picker

We can do much more with a date-picker control than just selecting a date from a pop-up calendar. We can disable date ranges, format the date, show it yearly and monthly, and so on. In this chapter, we will only learn how to get up and running with it.

To use a date-picker control, we first need to import the following modules:

- `MatDatepickerModule` from the `@angular/material/datepicker` namespace.
- `MatNativeDateModule` from the `@angular/material/core` namespace. It provides parsing and formatting utilities for dates, and it is based on the native `Date` object implementation.

A date-picker control in Angular Material must be used in conjunction with an input control, like the autocomplete control that we saw earlier:

```
<input matInput type="text" placeholder="Production date" />
```

The idea is that the input control triggers the date-picker control to be displayed. To create a date-picker control, we need to add a `<mat-datepicker-toggle>` element and a `<mat-datepicker>` element inside a `<mat-form-field>` element:

```
<mat-form-field>
  <input matInput type="text" placeholder="Production date" />
  <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
  <mat-datepicker #picker></mat-datepicker>
</mat-form-field>
```

The `<mat-datepicker-toggle>` element is a button with a calendar icon on it. It is positioned at the end of the input control, as defined by the `matSuffix` directive, and displays the calendar popup when clicked by the user. The `<mat-datepicker>` element defines a `picker` template reference variable that we can use to associate it both with the input field and the `<mat-datepicker-toggle>` element:

```
<input matInput type="text" placeholder="Production date"
[matDatepicker]="picker" />
```

You can see how a Datepicker looks in the product creation form when it is opened:

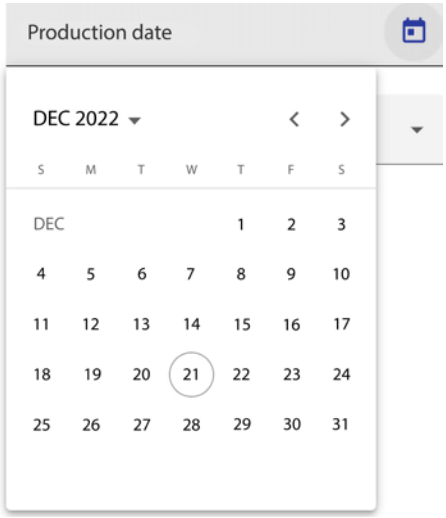


Figure 11.9: Production date control

The date picker is a form control used extensively in enterprise Angular applications. In the following section, we will learn about navigation techniques in Angular Material.

Navigation

There are different ways of navigating in an Angular application, such as clicking on 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 is a way for the user to reach commonly used actions.

In this section, we will demonstrate how to use the Toolbar component. However, we encourage you to keep exploring by learning how to use the menu and the sidenav components on the official Angular Material documentation website.

To create a toolbar for the main links of our e-shop application, we will go through the following steps:

1. Open the `app.module.ts` file and add the following `import` statement:

```
import { MatToolbarModule } from '@angular/material/toolbar';
```

2. Add the `MatToolbarModule` class in the `imports` array of the `@NgModule` decorator:

```
imports: [
  BrowserModule,
  BrowserAnimationsModule,
  MatButtonModule,
  MatIconModule,
  MatButtonToggleModule,
  ProductsModule,
  AppRoutingModule,
  HttpClientModule,
  AuthModule,
  ReactiveFormsModule,
  CommonModule,
  MatCheckboxModule,
  MatToolbarModule
]
```

3. Open the `app.component.html` file and modify the HTML content as follows:

```
<mat-toolbar color="primary">
  <span>My e-shop</span>
  <span class="spacer"></span>
  <a mat-flat-button color="primary" routerLink="/
products">Products</a>
  <a mat-flat-button color="primary" routerLink="/cart">Cart</a>
  <a mat-flat-button color="primary" routerLink="/about">About Us</
a>
  <app-auth></app-auth>
</mat-toolbar>
<router-outlet></router-outlet>
```

In the preceding template, we added the main application links and the authentication component inside a `<mat-toolbar>` element. The `<span>` element with the `spacer` class name is used to align content at the right end of the toolbar.

4. Open the `app.component.css` file and add the following CSS style:

```
.spacer {
  flex: 1 1 auto;
}
```

5. Open the `auth.module.ts` file and import `MatButtonModule`:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MatButtonModule } from '@angular/material/button';
import { AuthComponent } from './auth/auth.component';

@NgModule({
  declarations: [
    AuthComponent
  ],
  imports: [
    CommonModule,
    MatButtonModule
  ],
  exports: [
    AuthComponent
  ]
})
export class AuthModule { }
```

6. Now, open the `auth.component.html` file and add the `mat-raised-button` directive in the `Login` and `Logout` buttons:

```
<button mat-raised-button
  [hidden]="authService.isLoggedIn"
  (click)="authService.login().subscribe()"
>Login</button>
<button mat-raised-button
  [hidden]="!authService.isLoggedIn"
  (click)="authService.logout()"
>Logout</button>
```

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 11.10: Application toolbar

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.

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, and the component will fill out 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**: Works in a similar way to an accordion. It 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.

In the following sections, we will cover the list and grid-list components.

List

The list control is built up by a `<mat-list>` element that contains a set of `<mat-list-item>` elements:

```
<mat-list>
  <mat-list-item *ngFor="let product of products">
    {{product.name}}
  </mat-list-item>
</mat-list>
```

To use a `<mat-list>` element, we first need to import `MatListModule` from the `@angular/material/list` namespace. We can create simple lists, such as the previous snippet, or more advanced lists by enriching them with a multi-select functionality:

```
<mat-selection-list>
  <mat-list-option *ngFor="let product of products">
    {{product.name}}
```

```
    </mat-list-option>
  </mat-selection-list>
```

In the previous snippet, we use the `<mat-selection-list>` flavor of the list element that contains `<mat-list-option>` elements. A `<mat-list-option>` element is a list item with a label and a checkbox that we can check to select the item:

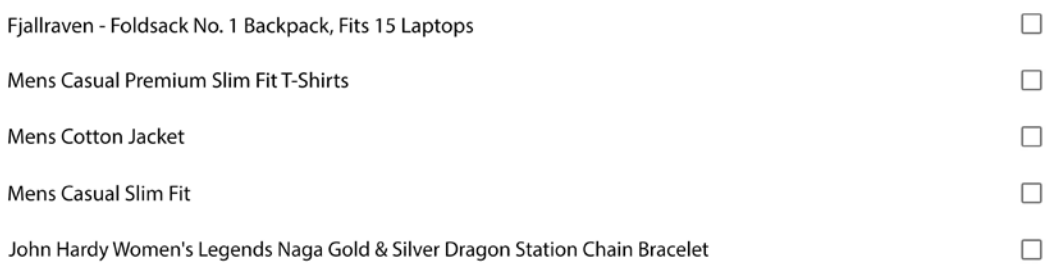


Figure 11.11: Selection list

The list control of the Angular Material library has a rich set of capabilities, and the combinations we can use are endless.

Grid list

A grid list is similar to a list control, but the content is arranged in a list of rows and columns while ensuring that it fills out the page viewport.

The viewport of a page is defined as the area of the page that is visible to the user. It varies according to the device that we use for browsing the content. For example, the viewport on mobile devices is smaller than on desktop.

It is an excellent fit if you want maximum freedom to decide how to display content. To use it, we must first import `MatGridListModule` from the `@angular/material/grid-list` namespace. The component consists of a `<mat-grid-list>` element and several `<mat-grid-tile>` elements:

```
<mat-grid-list cols="3" rowHeight="100px" gutterSize="50">
  <mat-grid-tile *ngFor="let product of products">
    {{product.name}}
  </mat-grid-tile>
</mat-grid-list>
```

We can set the number of columns and the height of each row by using the `cols` and `rowHeight` properties, respectively. We can also define the space between rows by setting the `gutterSize` property measured in pixels.

In the previous snippet, we use the `NgFor` directive to iterate over a list of products and display one tile for each one. The output should look like the following:

Figure 11.12: Grid list

The `<mat-grid-tile>` element also contains the following properties:

- `colspan`: Decides how many columns the tile should take
- `rowspan`: Indicates how many rows the tile should take

We encourage you to explore the preceding properties and the remaining card and tab components to learn more.

Popups and modal dialogs

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 about a part of the page when the user hovers over that part.

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.
- **Tooltip**: A piece of text that is displayed when we hover over a specific area.
- **Snackbar**: An information message displayed at the bottom of a page and is visible for a short time. Its purpose is to notify the user of the result of an action, such as saving a form.

In this chapter, we will focus on the dialog component, which is widely used in Angular applications. In the following section, we learn how to create a simple dialog.

Creating a simple dialog

The dialog component is quite powerful and can easily be customized and configured. It is an ordinary Angular component and uses custom directives that force it to behave like a dialog. To explore the capabilities of the Angular Material dialog, we will change the price of an existing product using a prompt dialog:

1. Open the `products.module.ts` file and add the following `import` statement:

```
import { MatDialogModule } from '@angular/material/dialog';
```

2. Add `MatDialogModule` in the `imports` array of the `@NgModule` decorator:

```
imports: [
  CommonModule,
  ProductsRoutingModule,
  FormsModule,
  ReactiveFormsModule,
  MatButtonModule,
  MatFormFieldModule,
  MatInputModule,
  MatAutocompleteModule,
  MatDatepickerModule,
  MatNativeDateModule,
  MatSelectModule,
  MatListModule,
  MatGridListModule,
  MatDialogModule
]
```

3. Run the following Angular CLI command in the `src\app\products` folder to create a new Angular component:

```
ng generate component price
```

The preceding command will create an Angular component that will be the host for our dialog.

4. Open the `price.component.ts` file and add a `price` property in the `PriceComponent` class:

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

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

```
export class PriceComponent {
  price: number | undefined;
}
```

5. Open the `price.component.html` file and replace its content with the following HTML template:

```
<h1 mat-dialog-title>Change product price</h1>
<mat-dialog-content>
  <mat-form-field>
    <input matInput [(ngModel)]="price" />
  </mat-form-field>
</mat-dialog-content>
<mat-dialog-actions>
  <button mat-raised-button color="primary" [mat-dialog-
close]="price">Save</button>
  <button mat-button mat-dialog-close>Cancel</button>
</mat-dialog-actions>
```

The component template contains various directives and elements that `MatDialogModule` exports and that we can use. The `mat-dialog-title` directive defines the title of the dialog, and `<mat-dialog-content>` is the actual content of the dialog. The `<mat-dialog-actions>` element defines the actions the dialog can perform and usually wraps button elements.

We use the `mat-dialog-close` directive on a button element to indicate that the dialog will be closed when that button is clicked. In our case, we use it twice. In the first case, we use it as a property binding and set it to a value that is finally passed back to the caller of the dialog. In the second case, we close the dialog right away.

6. A dialog must be triggered to be displayed on a page. Open the `product-detail.component.ts` file and add the following `import` statements:

```
import { MatDialog } from '@angular/material/dialog';
import { PriceComponent } from '../price/price.component';
```

7. Import the `filter` RxJS operator from the `rxjs` npm package:

```
import { filter, Observable, of, switchMap } from 'rxjs';
```

8. Inject the `MatDialog` service in the `constructor` of the `ProductDetailComponent` class:

```
constructor(
  private productService: ProductsService,
  public authService: AuthService,
  private route: ActivatedRoute,
  private cartService: CartService,
  private dialog: MatDialog
) { }
```

9. Modify the `changePrice` method as follows:

```
changePrice(product: Product) {
  this.dialog.open(PriceComponent).afterClosed().pipe(
    filter(price => !!price),
    switchMap(price => this.productService.updateProduct(product.id,
price))
  ).subscribe(() => {
    alert('The price of ${product.name} was changed!');
  });
}
```

   In the preceding method, we use the `MatDialog` service to display the `price` 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 that we can subscribe to, which will enable us to be notified when the dialog closes. The observable emits any value that is sent back from the dialog. Note that we check whether a value is returned from the dialog using the `filter` RxJS operator because the `Cancel` button does not return a value at all.

10. Finally, open the `product-detail.component.html` file and replace its content with the following HTML template:

```
<div *ngIf="product$ | async as product">
  <h2>Product Details</h2>
  <h3>{{product.name}}</h3>
  <span>{{product.price | currency:'EUR'}}</span>
  <button mat-raised-button color="primary"
(click)="changePrice(product)">Change</button>
```

```
    <p>
      <button mat-button *ngIf="authService.isLoggedIn"
(click)="buy(product)">Buy Now</button>
      <button mat-raised-button color="primary" class="delete"
(click)="remove(product)">Delete</button>
    </p>
</div>
```

Run the application using the `ng serve` command, select a product from the list, and click the **Change** button. The following dialog will appear on the screen:

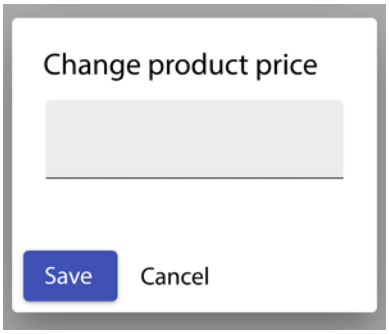


Figure 11.13: Change price dialog

You may have noticed that the input field for entering the new price is empty. In the following section, we will learn how to pass data in the dialog and display the current price as a placeholder inside the input control.

Configuring a dialog

In a real-world scenario, you will probably need to create a reusable component for displaying a dialog in an Angular project. Even better, 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 current price of the product when the user wants to change it using the price dialog:

1. Open the `price.component.ts` file and import `MAT_DIALOG_DATA` from the `@angular/material/dialog` namespace and `Inject` from the `@angular/core` library:

```
import { Component, Inject } from '@angular/core';
import { MAT_DIALOG_DATA } from '@angular/material/dialog';
```

2. Inject `MAT_DIALOG_DATA` in the `constructor` of the `PriceComponent` class in the following way:

```
export class PriceComponent {

  price: number | undefined;

  constructor(@Inject(MAT_DIALOG_DATA) public data: number) {}

}
```

   The `MAT_DIALOG_DATA` is not an Angular service, so we cannot inject it normally as we do with services. It is an injection token, and we use the `@Inject` decorator to inject it, as we learned in *Chapter 6*, *Managing Complex Tasks with Services*. The `data` variable will contain any data we pass to the dialog when we call its open method.

3. Open the `price.component.html` file and bind the `data` property to the `placeholder` property of the `<input>` element:

```
<input matInput [(ngModel)]="price" [placeholder]="data.toString()"
/>
```

4. Open the `product-detail.component.ts` file and set the `data` property in the dialog configuration object, which is the second parameter of the open method:

```
changePrice(product: Product) {
  this.dialog.open(PriceComponent, {
    data: product.price
  }).afterClosed().pipe(
    filter(price => !!price),
    switchMap(price => this.productService.updateProduct(product.id,
price))
  ).subscribe(() => {
    alert('The price of ${product.name} was changed!');
  });
}
```

We have already seen how to return data from the dialog to the caller component using the `mat-dialog-close` directive. In the following section, we will learn how to accomplish the same task programmatically from the component class.

Getting data back from a dialog

Instead of using the `mat-dialog-close` directive to close a dialog declaratively, we could use the `MatDialogRef` service. `MatDialogModule` exports the `MatDialogRef` service that contains a `close` method we can use in the `price.component.ts` file:

```
import { Component, Inject } from '@angular/core';
import { MatDialogRef, MAT_DIALOG_DATA } from '@angular/material/dialog';

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

  price: number | undefined;

  constructor(
    @Inject(MAT_DIALOG_DATA) public data: number,
    private dialogRef: MatDialogRef<PriceComponent>
  ) {}

  save() {
    this.dialogRef.close(this.price);
  }

}
```

The `close` method accepts a single parameter that defines the data we want to send back to the caller.

When we inject the `MatDialogRef` service, we also set its type to `PriceComponent`, the same as the dialog component itself.

We should also modify the `Save` button in the `price.component.html` file accordingly:

```
<button mat-raised-button color="primary" (click)="save()">Save</button>
```

Dialogs are a great feature of Angular Material that can give powerful capabilities to your Angular applications. In the following section, we will explore how to display tabular data in your Angular applications with data tables.

Data table

We can visualize data in an Angular component in different ways. An efficient way of getting a quick overview is by displaying it in a tabular format with rows and columns. However, we might need to sort data by column to find the information we are looking for. Also, the amount of data might be so large that it needs to be shown in parts by page. Angular Material addresses all these issues by offering the following components:

- **Table**: Lays out data in rows and columns with headers
- **Sort table**: Allows you to sort data in a table
- **Paginator**: Allows you to slice up data in pages that we can navigate

In the following sections, we learn more about each component in detail

Table

The table component allows us to display our data in columns and rows. To create a table, we first need to import `MatTableModule` from the `@angular/material/table` namespace.

An Angular Material table is a standard HTML `<table>` element that contains specific Angular directives to conform to the Material Design guidelines. To create the table initially, we use the `mat-table` directive:

```
<table mat-table [dataSource]="products"></table>
```

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` array that we declared in our component class, along with the `columnNames` property, which indicates the column names of the table:

```
columnNames = ['name', 'price'];
```

The names of columns match the properties of a `Product` object and are used twice in the table element—once to define the header row of the table that displays the names of the columns and the second time to define the actual rows that contain data:

```
<tr mat-header-row *matHeaderRowDef="columnNames"></tr>
<tr mat-row *matRowDef="let row; columns: columnNames;"></tr>
```

Finally, we use an `<ng-container>` element for each column to display the header and data cells:

```
<ng-container matColumnDef="name">
  <th mat-header-cell *matHeaderCellDef>Name</th>
  <td mat-cell *matCellDef="let product">
    <a [routerLink]="['/products', product.id]">{{product.name}}</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>
```

The `<ng-container>` element is a unique-purpose element used to group 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, as defined in the `columnNames` component property.

The value of `matColumnDef` 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>` element with a `mat-header-cell` directive that indicates the header of the cell and a `<td>` element with a `mat-cell` directive for the data of the cell. The `<td>` element uses the `matCellDef` directive to create a local template variable for the current row data we can bind to later.

If we run the application, the output should be the following:

Name	Price
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops	$109.95
Mens Casual Premium Slim Fit T-Shirts	$22.30
Mens Cotton Jacket	$55.99
Mens Casual Slim Fit	$15.99
John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet	$695.00
Solid Gold Petite Micropave	$168.00
White Gold Plated Princess	$9.99
Pierced Owl Rose Gold Plated Stainless Steel Double	$10.99
WD 2TB Elements Portable External Hard Drive - USB 3.0	$64.00

Figure 11.14: Table control

You did great! You managed to create a beautiful table in no time using Angular Material.

Sort table

At this point, we have created a nice-looking table, but it lacks a pretty standard functionality—sorting. We would typically expect that if we click the header, it will sort data into ascending and descending order, respectively, and it will be able to recognize common data types, such as text and numbers, and sort them properly. The good news is that Angular Material can help us to achieve this behavior.

We need to use the appropriate Angular Material directives for the job:

1. Open the `products.module.ts` file and import `MatSortModule` from the `@angular/material/sort` namespace. Add it also in the `imports` array of the `@NgModule` decorator.
2. `MatSortModule` exports a variety of directives that we can use to sort a table. Open the `product-list.component.html` file and add the `matSort` and `matSortDisableClear` directives to the `<table>` element and the `mat-sort-header` directive to each header cell:

```
<table mat-table [dataSource]="products" matSort
matSortDisableClear>
  <ng-container matColumnDef="name">
    <th mat-header-cell *matHeaderCellDef mat-sort-header>Name</th>
    <td mat-cell *matCellDef="let product">
      <a [routerLink]="['/products', product.id]">{{product.name}}</
a>
    </td>
  </ng-container>
  <ng-container matColumnDef="price">
    <th mat-header-cell *matHeaderCellDef mat-sort-header>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>
```

   In the preceding snippet, we add the `matSortDisableClear` directive because sorting by default contains three states: ascending, descending, and the original ordering. The last one clears the ordering, which is why we disable it.

3. Use the `@ViewChild` decorator inside the `ProductListComponent` class to get a reference to the `matSort` directive that we defined earlier:

```
@ViewChild(MatSort) sort: MatSort | null = null;
```

4. To use sorting, wrap the data in a `MatTableDataSource` instance and set the `sort` property of that instance to the `MatSort` property that we defined previously:

```
private getProducts() {
  this.productService.getProducts().subscribe(products => {
    this.products = new MatTableDataSource(products);
```

```
      this.products.sort = this.sort;
    });
  }
```

Sorting a table is a feature you may need when writing Angular applications. The sorting configuration looks simple as soon as you have a simple table model.

Pagination

So far, our table is starting to look quite good. As well as displaying data, it can even be sorted. We are aware, though, that in most cases, the data for a table is usually quite long, which means that the user either has to scroll up and down or browse the data page by page. We can solve the latter problem with the help of the pagination element. To use it, we need to do the following:

1. Open the `products.module.ts` file and import `MatPaginatorModule` from the `@angular/material/paginator` namespace. Add it also in the `imports` array of the `@NgModule` decorator.
2. Open the `product-list.component.html` file and add a `<mat-paginator>` element immediately after the `<table>` element. Set the `pageSize` property to display five rows each time. Also, set the `pageSizeOptions` property so that the user can change the page size:

```
<mat-paginator [pageSize]="5" [pageSizeOptions]="[5,10,15]"></mat-paginator>
```

3. Open the `product-list.component.ts` file and use the `@ViewChild` decorator inside the `ProductListComponent` class to get a reference to the `<mat-paginator>` element that we created:

```
@ViewChild(MatPaginator) paginator: MatPaginator | null = null;
```

4. Set the `paginator` property of the `products` property to the `MatPaginator` property that we defined previously:

```
private getProducts() {
  this.productService.getProducts().subscribe(products => {
    this.products = new MatTableDataSource(products);
    this.products.sort = this.sort;
    this.products.paginator = this.paginator;
  });
}
```

If we run the application, we will notice that the table now displays five products at a time; however, we can navigate through all of the pages using the paginator control that is shown at the bottom of the table:

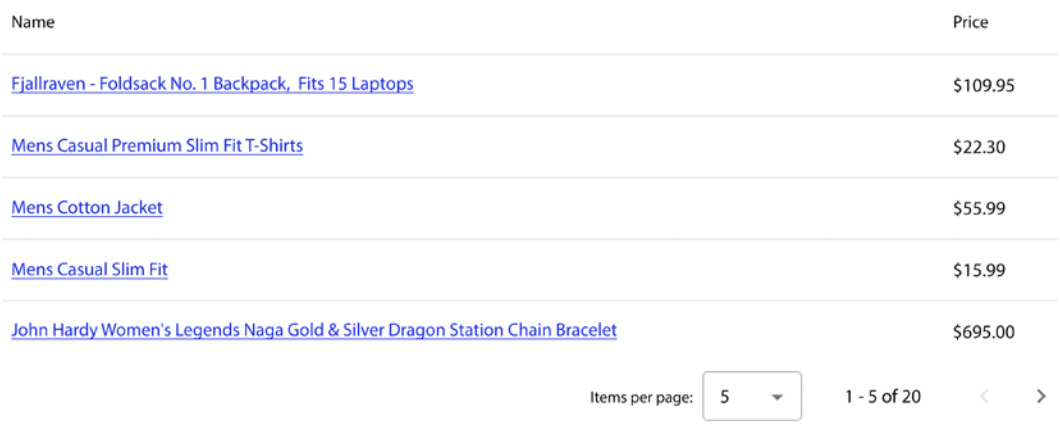


Figure 11.15: Table with pagination

The `paginator` component also displays the total length of our data, even if we did not set it explicitly. Well, we did when we set the `paginator` property of the data source to the paginator element. It is smart enough to understand how to handle the data by itself.

In this section, we learned about some of the core components of the Angular Material library and how we can leverage them to create compelling and engaging user interfaces. We covered UI controls that span various uses, such as navigation, layout, popups, and form controls.

In the next section, we will learn how to integrate other Google products in an Angular application, such as the YouTube player and Google Maps.

Integration controls

The Angular Material library contains two special-purpose controls that integrate external Google products into an Angular application: Google Maps and YouTube.

To start using the Google Maps control, we first need to install it from npm as a separate library:

```
npm install @angular/google-maps
```

In this book we work with Angular Material 15 which is compatible with Angular 15. If the version of Google Maps that you installed is different, you should run the command `npm install @angular/google-maps@15` to install the latest Google Maps 15 to your system.

The Google Maps control works properly only if we import the Maps JavaScript API in an Angular application. To import the API, we need to add the following snippet in the `<head>` element of the `index.html` file replacing the `YOUR_API_KEY` variable with a valid Maps API key:

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY"></
script>
```

The components we need for creating a map in an Angular application are exposed in `GoogleMapsModule`, which is part of the `@angular/google-maps` npm package and must be imported into an Angular module of our application.

To display an instance of Google Maps in our Angular application, walk through the following steps:

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

   ```
   ng generate component map
   ```

2. Open the `map.component.ts` file and create the following properties in the `MapComponent` class:

   ```
   position: google.maps.LatLngLiteral =  {
     lat: 38.480052,
     lng: 22.494062
   };
   options: google.maps.MapOptions = {
     center: { lat: 39.0742, lng: 21.8243 },
     zoom: 6
   };
   ```

   In the preceding snippet, the `options` property will be used to define the map center and the zoom level. The `position` property will be used to locate a map marker in the specified latitude and longitude coordinates inside the map.

3. Open the `map.component.html` file and replace its content with the following HTML template:

   ```
   <google-map [options]="options">
     <map-marker [position]="position"></map-marker>
   </google-map>
   ```

In the preceding template, we use the `<google-map>` element to define a map control with specific `options`. Inside the map, we add a `<map-marker>` element to add a map marker in the specified position.

The map we created is a basic one with a single marker. The Google Maps API in Angular Material has much more capabilities that we encourage you to check out.

Another external application that we can embed in an Angular application is the YouTube player. To start using it, we must first install it from the npm registry with the following command:

```
npm install @angular/youtube-player
```

In this book we work with Angular Material 15 which is compatible with Angular 15. If the version of the YouTube player that you installed is different, you should run the command `npm install @angular/youtube-player@15` to install the latest YouTube player 15 to your system.

The Angular Material component used to add YouTube player in an Angular component is available from the `YouTubePlayerModule` class. It can be imported from the `@angular/youtube-player` npm package and should be imported into an Angular module of our application.

We can then use the `<youtube-player>` element, passing a video ID as a parameter:

```
<youtube-player videoId="YOUR_VIDEO_ID"></youtube-player>
```

The ID of the video can be found in the `v` parameter of a YouTube video URL, such as `https://www.youtube.com/watch?v=VoIDrdjgpR8`.

Finally, we add the `ngOnInit` method in the `app.component.ts` file to load the YouTube API *once*:

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

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

```
  ngOnInit(): void {
    const tag = document.createElement('script');
    tag.src = 'https://www.youtube.com/iframe_api';
    document.body.appendChild(tag);
  }

}
```

In the next section, we will learn about the backbone of the Angular Material library, the Angular CDK, and how we can use it to create custom controls that adhere to Material Design guidelines.

Introducing the Angular CDK

The Angular CDK is the core of the Angular Material library. It is a collection of tools that implement similar interaction patterns; however, they are not tied to any presentation style, such as Material Design. 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.

The capabilities of the Angular CDK are enormous and certainly cannot fit in a single chapter. For the sake of demonstration, we are going to describe two elements of the library:

- **Clipboard**: Provides a copy-paste functionality with the system clipboard
- **Drag and drop**: Provides drag-and-drop features in elements

Angular CDK elements are imported from the `@angular/cdk` npm package. Each element must be imported from its module, which resides in a different namespace, similar to the Angular Material components.

Clipboard

We can easily create a *copy-to-clipboard* button using the `cdkCopyToClipboard` directive. All we have to do is import `ClipboardModule` from the `@angular/cdk/clipboard` namespace and attach the directive to a button element, such as in the following HTML template:

```
<mat-form-field>
  <textarea matInput [(ngModel)]="content" placeholder="Enter some text
and click the Copy button"></textarea>
</mat-form-field>
<button mat-flat-button [cdkCopyToClipboard]="content">
```

```
  <mat-icon>content_copy</mat-icon>
  Copy
</button>
```

We set the value of the directive to the `content` property in the respective component class:

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

@Component({
  selector: 'app-copy-text',
  templateUrl: './copy-text.component.html',
  styleUrls: ['./copy-text.component.css']
})
export class CopyTextComponent {
  content = '';
}
```

It is the actual content that is going to be copied to the clipboard once we click the **Copy** button.

Drag and drop

A powerful application of the drag-and-drop functionality is when using lists in an Angular application, which we do in most cases! To use it, we must first import `DragDropModule` from the `@angular/cdk/drag-drop` namespace. The drag-and-drop component of Angular CDK is spread across various directives that we can apply to a `<mat-list>` element:

```
<mat-list cdkDropList>
  <mat-list-item cdkDrag *ngFor="let product of products">
    {{product.name}}
  </mat-list-item>
</mat-list>
```

The `cdkDropList` directive indicates that the `<mat-list>` element is a container for items that can be dragged. The `cdkDrag` directive indicates that the `<mat-list-item>` element can be dragged. We have also applied a bit of styling to identify the items as draggable:

```
mat-list-item {
  cursor: move;
  border: 1px lightgray solid;
}
```

If we run the application using the `ng serve` command, we will notice that even if we can drag an item from the list, the application will not respect the movement of the item when we drop it. The drag-and-drop component does not have reordering baked in, but we must implement it on our own. We can use the `cdkDropListDropped` event binding to achieve that:

```
<mat-list cdkDropList (cdkDropListDropped)="reorder($event)">
  <mat-list-item cdkDrag *ngFor="let product of products">
    {{product.name}}
  </mat-list-item>
</mat-list>
```

When we drag a `<mat-list-item>` element and drop it, the `reorder` component method will be called, as defined in the `products.component.ts` file:

```
reorder(event: CdkDragDrop<Product[]>) {
  moveItemInArray(this.products, event.previousIndex, event.currentIndex);
}
```

It accepts a `CdkDragDrop` event of the `Product[]` type. Although the Angular CDK cannot reorder items by itself, it gives us the necessary artifacts to perform reordering efficiently. We use the built-in `moveItemInArray` method from the `@angular/cdk/drag-drop` namespace, which performs reordering out of the box. It accepts three parameters: the array we want to sort, the index of the current item that we drag it from, and the new index that we will drop in.

Summary

In this chapter, we looked at Material Design, a design language with paper and ink in mind.

Next, 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. We also learned about the core of Angular Material, the Angular CDK, and demonstrated some of its style-aware components.

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 the context of Angular web applications.

12

Unit Test an Angular Application

In the previous chapters, we have gone 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. However, we will briefly cover all the intricacies of testing an Angular application in this chapter now that our knowledge of the framework is mature.

In this chapter, we will learn how to use testing tools to perform proper unit testing of our Angular application artifacts. In more detail, we will learn about the following:

- Why do we need tests?
- The anatomy of a unit test
- Introducing unit tests in Angular
- Testing components
- Testing services
- Testing pipes
- Testing directives
- Testing forms

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 `ch12` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`.

Why do we need tests?

What is a unit test? You can skip to the next section if you're already familiar with unit testing and test-driven development. If not, let's say that 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 the daily development workflow. Why is that? Building tests while 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 the other way around. If we try to test an existing piece of code and we 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 with every refactor, the risk of introducing a bug is high. Unit tests are an excellent way to ensure that we catch bugs at an early stage, 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 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 test—the different parts it's made of. To test any code, we need a framework for writing the test and a runner to run it on.

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**, a popular test framework, which is also used by default in Angular CLI projects. Here is how a unit test looks in Jasmine:

```
describe('Calculator', () => {
  it('should add two numbers', () => {
    expect(1+1).toBe(2);
  });
});
```

The `describe` method defines a 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, 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 that the previous code contains another mathematical operation that needs to be tested. It would make sense to group both operations under one suite:

```
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. There are, however, more concepts involved in unit tests that are worth knowing about, namely the setup and teardown functionalities.

A setup functionality is something that prepares your code before you start running the tests. It's a way to keep your code clean to focus on invoking the code and checking the assertions. A teardown functionality is the opposite of a setup functionality. It is responsible for tearing down what we initially set up, which is involved in 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 tear-down 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 going on in a real-world application. Most importantly, the `beforeEach` method tends to make it easier to add new tests, which is great. At the end of the day, 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 Angular framework context.

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 to make unit testing a pleasant experience:

- **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 as soon as 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 that can be found under 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 is the most crucial concept. It essentially creates a testing module that behaves like an ordinary Angular module. When we test an Angular artifact, we detach it from the Angular module it resides in and attach it to this testing module. 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. We will embark on this great journey from the most fundamental building block in Angular, the component.

Testing components

You may have noticed that every time 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 that it is testing, followed by the suffix `.spec.ts`. For example, the test file for the main component of an Angular application, `app.component.ts`, would be `app.component.spec.ts` and would reside in the same path as the component file.

We should think about an Angular artifact and its corresponding test as 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 do some refactoring to 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({
    declarations: [
      AppComponent
    ],
  }).compileComponents();
});
```

It uses the `configureTestingModule` method of the `TestBed` class and passes an object as a parameter. The properties of the object are almost the same as those of the `@NgModule` decorator.

We can take our knowledge of configuring an Angular module and apply that to set up a testing module. We can specify a `declarations` array that contains the component we want to test. Additionally, we can define tear-down options for the module 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 any bugs that happen due to the incomplete cleanup of HTML elements.
- `rethrowErrors`: It throws any errors that occur when the module is destroyed.

Finally, we call the `compileComponents` method, and the setup is completed.

The `compileComponents` method, as per its name, compiles components configured in the testing module. During the compilation process, it inlines external CSS files and templates. We will not use this method for the rest of this chapter because the Angular CLI does it for us under the hood; however, do not forget that Angular testing utilities can be used with build tools other than the Angular CLI.

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 as title 'my-app'', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.title).toEqual('my-app');
});
```

In the previous test, we checked whether the `title` component property is set to `my-app` using another matcher function, `toEqual`.

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('.content span')?.textContent).
toContain('my-app app is running!');
});
```

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 similar to what we did before, and we 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 that corresponds 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:

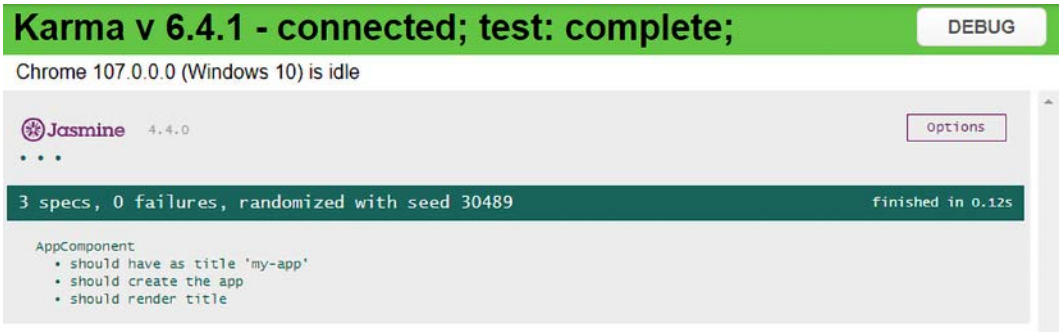


Figure 12.1: Test execution output

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:

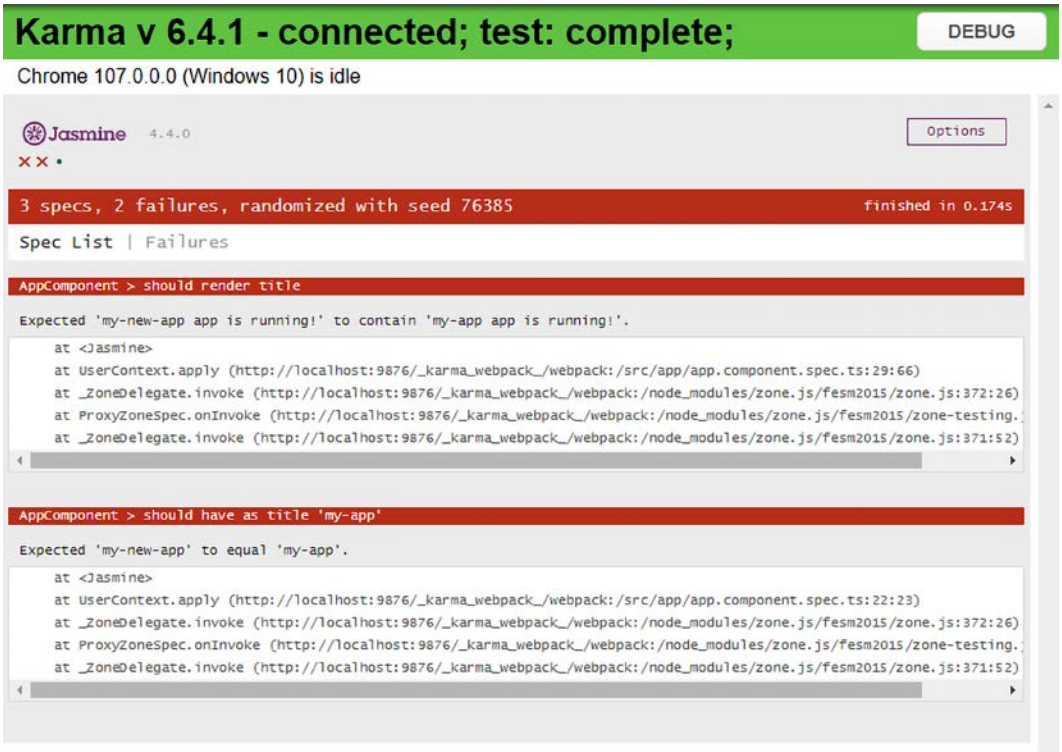


Figure 12.2: Test failure

Karma runs in watch mode, so we do not need to execute the Angular CLI test command every time we make a change.

In some cases, it is not very convenient to read the output of tests in the browser. 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:

```
Chrome 107.0.0.0 (Windows 10): Executed 3 of 3 SUCCESS (0.134 secs / 0.122 secs)
TOTAL: 3 SUCCESS
```

Figure 12.3: Console test output

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 `AppComponent`. They will almost certainly be dependent on one or more services. We have different ways of dealing with testing in such a situation. One thing is clear, though: if we are testing the component, then we should not test the service as well. So when we set up such a test, the dependency should not be the real thing. There are different ways of dealing with that when it comes to unit testing; no solution is strictly better than another:

- **Stubbing**: This is the method of telling the dependency injector to inject a stub of the dependency that we provide instead of the real thing.
- **Spying**: This is the method of injecting the actual dependency but attaching 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 into their constructor, so using the real dependency in a test requires you to compensate for other dependencies.

Regardless of the approach, we ensure that the test does not perform unintended actions, such as talking to a 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 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. 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 stub: StubService) { }

  ngOnInit() {
    this.msg = !this.stub.isBusy
      ? this.stub.name + ' is available'
      : this.stub.name + ' is on a mission';
  }

}
```

It injects `StubService`, which contains just two public properties. Providing a stub for this service 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({
  declarations: [ StubComponent ],
  providers: [
    { provide: StubService, useValue: serviceStub }
  ]
});
```

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:

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [ StubComponent ],
    providers: [
      { provide: StubService, useValue: serviceStub }
    ]
  });

  fixture = TestBed.createComponent(StubComponent);
  component = fixture.componentInstance;

  msgDisplay = fixture.nativeElement.querySelector('span');
  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', () => {
  it('should be on a mission', () => {
    service.isBusy = true;
    fixture.detectChanges();
    expect(msgDisplay.textContent).toContain('is on a mission');
  });

  it('should be available', () => {
    service.isBusy = false;
    fixture.detectChanges();
    expect(msgDisplay.textContent).toContain('is available');
  });
});
```

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 saw earlier doesn't have any effect. So how do we tackle such a scenario? We use the `overrideComponent` method of the `TestBed` class:

```
await TestBed.configureTestingModule({
  declarations: [ StubComponent ],
})
.overrideComponent(StubComponent, {
  set: {
    providers: [
      { provide: StubService, useValue: serviceStub }
    ]
  }
});
```

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 is used to provide services to the component.

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

The previously mentioned approach, 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 it to answer, 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 instance of the dependency. We can then spy on the methods of this dependency as we would with the real one.

Let's see how to set up the first case. 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() {
    this.title.setTitle('My Angular app');
    this.caption = this.title.getTitle();
  }

}
```

The `Title` service is used to interact with the title of the HTML document of an Angular application and can be imported from the `@angular/platform-browser` npm package.

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. The easiest and safest way to use it in our tests is by spying on its methods. We inject it in the testing module using the `providers` array of the `@NgModule` decorator and then use it in our test like this:

```
it('should set the title', () => {
  const title = TestBed.inject(Title);
  const spy = spyOn(title, 'setTitle');
  fixture.detectChanges();
  expect(spy).toHaveBeenCalledWith('My Angular app');
});
```

We use the Jasmine `spyOn` method, which accepts two parameters: the object and its specific method to spy. Note that we use it before calling the `detectChanges` method since we want to attach the spy before triggering the `ngOnInit` life cycle hook. The `expect` statement then validates that the `setTitle` method was called with the correct argument.

Our component also uses another method of the `Title` service—the `getTitle` method—to get the document title. We can leverage the second case, which we defined before, to spy on the method and return mock data:

1. First, we need to define the `Title` service as a spy object:

   ```
   let titleSpy: jasmine.SpyObj<Title>;
   ```

2. Then, we use the `createSpyObj` method to initialize the spy object, passing two parameters: the name of the service and an array of the method names that the component currently uses:

   ```
   titleSpy = jasmine.createSpyObj('Title', ['getTitle', 'setTitle']);
   ```

3. Finally, we attach a spy to the `getTitle` method and return a custom title using the Jasmine `returnValue` method:

   ```
   titleSpy.getTitle.and.returnValue('My title');
   ```

As soon as we add the `titleSpy` variable in the `providers` array of the testing module, we can use it in our tests. The resulting test suite should look like the following:

```
describe('with spy object', () => {
  let titleSpy: jasmine.SpyObj<Title>;

  beforeEach(() => {
    titleSpy = jasmine.createSpyObj('Title', ['getTitle', 'setTitle']);
    titleSpy.getTitle.and.returnValue('My title');

    TestBed.configureTestingModule({
      declarations: [ SpyComponent ],
      providers: [
        { provide: Title, useValue: titleSpy }
      ]
    });
    fixture = TestBed.createComponent(SpyComponent);
    component = fixture.componentInstance;
  });

  it('should get the title', () => {
    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 asynchronous services. It is combined with the `whenStable` method of `ComponentFixture`.
- `fakeAsync`: A synchronous approach to unit test asynchronous 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 { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { AsyncService } from '../async.service';

@Component({
  selector: 'app-async',
  template: '
    <p *ngFor="let hero of data$ | async">
      {{hero}}
    </p>
  '
})
export class AsyncComponent implements OnInit {
  data$: Observable<string[]> | undefined;

  constructor(private asyncService: AsyncService) { }

  ngOnInit() {
    this.data$ = this.asyncService.getData();
  }
}
```

It injects the `AsyncService` from the `async.service.ts` file and calls its `getData` method inside the `ngOnInit` method. As we can see, the `getData` method returns an observable of strings. It also introduces a slight delay so that the scenario looks asynchronous:

```
getData(): Observable<string[]> {
  return of(heroes).pipe(delay(500));
}
```

The unit test queries the native element of the component and checks whether the NgFor directive loops through the `data$` observable correctly:

```
it('should get data with waitForAsync', waitForAsync(async() => {
  fixture.detectChanges();
  await fixture.whenStable();
```

```
    fixture.detectChanges();
    const heroDisplay: HTMLElement[] = fixture.nativeElement.
querySelectorAll('p');
    expect(heroDisplay.length).toBe(5);
}));
```

We wrap the body of the test inside the `waitForAsync` method, and initially, we call the `detectChanges` method to trigger the `ngOnInit` lifecycle hook. Furthermore, we call the `whenStable` method, which returns a promise, which is resolved immediately when the `data$` observable is complete. When the promise is resolved, we call `detectChanges` once more 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 preferrable 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 data with fakeAsync', fakeAsync(() => {
  fixture.detectChanges();
  tick(500);
  fixture.detectChanges();
  const heroDisplay: HTMLElement[] = fixture.nativeElement.
querySelectorAll('p');
  expect(heroDisplay.length).toBe(5);
}));
```

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, the virtual delay we introduced in the `getData` method of `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 4*, *Enabling User Experience with Components*, a component has a public API consisting of inputs and outputs that should be tested as well.

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, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-bindings',
  template: '
    <p>{{title}}</p>
    <button (click)="liked.emit()">Like!</button>
  '
})
export class BindingsComponent {
  @Input() title = '';
  @Output() liked = new EventEmitter();
}
```

Before we start writing our tests, we should create a test host component that is going to use the component under test:

```
@Component({
  template: '<app-bindings [title]="testTitle" (liked)="isFavorite =
true"></app-bindings>'
})
export class TestHostComponent {
  testTitle = 'My title';
  isFavorite = false;
}
```

In the setup phase, we declare both components in the testing module but notice that the `ComponentFixture` is of the `TestHostComponent` type:

```
let component: TestHostComponent;
let fixture: ComponentFixture<TestHostComponent>;

beforeEach(async () => {
  await TestBed.configureTestingModule({
    declarations: [
      BindingsComponent,
      TestHostComponent
    ]
  });

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

We follow this approach because we want to test `BindingsComponent` when used with a host component, not by itself. 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.testTitle);
});
```

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 `ComponentFixture` and then 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`, and it 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 we learned about in *Chapter 11*, *Introduction to 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 by 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 built-in testing environments for unit testing with Karma and end-to-end testing with **Protractor**. The Angular CDK also provides a rich set of tools that allow developers to create custom testing environments.
- **Harness loader**: A class used to load a component harness inside a unit test.
- **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, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-bindings',
  template: '
    <p>{{title}}</p>
    <button mat-button (click)="liked.emit()">Like!</button>
  '
})
export class BindingsComponent {
  @Input() title = '';
  @Output() liked = new EventEmitter();
}
```

We have already learned how to add the Angular Material library and use the button component in *Chapter 11*, *Introduction to Angular Material*. 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 { HarnessLoader } from '@angular/cdk/testing';
```

The `TestbedHarnessEnvironment` class represents the testing environment for running unit tests with Karma. The `HarnessLoader` class will be used later to create a harness loader instance. We also need to add the following `import` statement:

```
import { MatButtonHarness } from '@angular/material/button/testing';
```

The `MatButtonHarness` class is 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 use the `TestbedHarnessEnvironment` to create a harness loader:

```
loader = TestbedHarnessEnvironment.loader(fixture);
```

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.

After we have created a harness loader, we can use it to start writing our unit tests:

```
it('should emit the liked event using harness', async () => {
  const buttonHarness = await loader.getHarness(MatButtonHarness);
  await buttonHarness.click();
  expect(component.isFavorite).toBeTrue();
});
```

In the preceding test, 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 harnesses trigger change detection automatically.

The component harness is a powerful Angular CDK tool that ensures we interact with components during testing in an abstract and safe way.

We have gone through many ways to test a component with a dependency. Now it is time to learn how to test the dependency by itself.

Testing services

As we learned in *Chapter 6*, *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 symbols do not have any value in being tested because they represent the internal implementation of the service.

There are three different types of testing that we can perform in a service:

- Testing a synchronous operation, such as a method that returns a simple array
- Testing an asynchronous operation, such as a method 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 of them in more detail.

Testing a synchronous method

When we create an Angular service using the Angular CLI, it also creates a corresponding test file. When we created the `async` service, the Angular CLI created the following `async.service.spec.ts` file:

```
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 initially 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 that we test using the `inject` method of the `TestBed` class.

When a service is provided from an injector other than the root, we should add it to the `providers` array of the testing module, as we did with the components.

The first test that we can write is pretty straightforward as it calls the `setData` method and inspects its result:

```
it('should set data', () => {
  const result = service.setData('Fake hero');
  expect(result.length).toBe(6);
});
```

Writing a test for synchronous methods is usually relatively easy; however, things are different when we want to test an asynchronous method.

Testing an asynchronous method

The second test is a bit tricky because it involves an observable. We need to subscribe to the `getData` method and inspect the value as soon as the observable is complete:

```
it('should get data', (done: DoneFn) => {
  service.getData().subscribe(heroes => {
    expect(heroes.length).toBe(5);
    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 different method we saw in the *Testing components* section can be applied in the same way; however, we follow a different approach when testing a service that injects the `HttpClient` service.

Consider the following methods from the `products.service.ts` file that we have already used in previous chapters:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    }))
  );
}

addProduct(name: string, price: number): Observable<Product> {
  return this.http.post<ProductDTO>(this.productsUrl, {
    title: name,
    price: price
  }).pipe(
    map(product => this.convertToProduct(product))
  );
}
```

Angular testing utilities provide two artifacts for mocking HTTP requests in unit tests: the `HttpClientTestingModule`, which replaces the real `HttpClientModule`, and the `HttpTestingController`, which mocks the `HttpClient` service. We can import both from the `@angular/common/http/testing` namespace:

```
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule]
  });
  service = TestBed.inject(ProductsService);
  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 `getProducts` method:

```
it('should get products', () => {
  service.getProducts().subscribe();
  const req = httpTestingController.expectOne('https://fakestoreapi.com/
products');
```

```
  expect(req.request.method).toBe('GET');
});
```

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 `addProduct` method, except that we need to make sure that the body of the request contains the correct data:

```
it('should add a product', () => {
    service.addProduct('Fake product', 100).subscribe();
    const req = httpTestingController.expectOne('https://fakestoreapi.com/
products');
    expect(req.request.method).toBe('POST');
    expect(req.request.body).toEqual({
      title: 'Fake product',
      price: 100
    });
  });
```

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 5*, *Enrich 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. The `list.pipe.ts` file contains 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 for it is simple. The only thing that we need to do is to instantiate an instance of `ListPipe` and verify the outcome of the `transform` method with some mock data:

```
import { ListPipe } from './list.pipe';

describe('ListPipe', () => {
  it('create an instance', () => {
    const pipe = new ListPipe();
    expect(pipe).toBeTruthy();
  });

  it('should return an array', () => {
    const pipe = new ListPipe();
    expect(pipe.transform('A,B,C')).toEqual(['A', 'B', 'C']);
  });
});
```

It is worth noting that Angular testing utilities are not involved when testing a pipe. We create an instance of the pipe class, and we can start calling methods. Pretty simple!

Angular directives are Angular 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. We will learn how to accomplish this task in the following section.

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 very good idea of how to proceed when testing them.

Consider the `copyright.directive.ts` file that we created in *Chapter 5*, *Enrich 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 in conjunction 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 `declarations` array of the testing module along with the directive under test:

```
@Component({
  template: '<span appCopyright></span>'
})
class TestHostComponent { }
```

We can now write our tests that check whether the `<span>` element contains the `copyright` class and displays the current year in its `textContent` property:

```
describe('CopyrightDirective', () => {
  let container: HTMLElement;

  beforeEach(() => {
    const fixture = TestBed.configureTestingModule({
      declarations: [
        CopyrightDirective,
        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.

We will end our testing journey by looking at how to test Angular forms.

Testing forms

As we saw in *Chapter 10*, *Collecting User Data with Forms*, forms are an integral part of an Angular application. It is rare for an Angular application not to at least have a simple form, such as a search form. We have already learned that reactive forms are better than template-driven forms in many ways and are easier to test, so in this section, we will focus on only testing reactive forms.

Consider the following `search.component.ts` file:

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

@Component({
  selector: 'app-search',
  template: '
    <form [formGroup]="searchForm" (ngSubmit)="search()">
      <input type="text" placeholder="Username"
formControlName="searchText">
```

```
      <button type="submit" [disabled]="searchForm.invalid">Search</
button>
    </form>
  '
})
export class SearchComponent {

  get searchText() {
    return this.searchForm.controls.searchText;
  }
  searchForm = new FormGroup({
    searchText: new FormControl('', Validators.required)
  });

  search() {
    if(this.searchForm.valid) {
      console.log('You searched for: ' + this.searchText.value)
    }
  }
}
```

In the preceding component, we can write our unit tests to verify that:

- The `searchText` property 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, as we would in an Angular module:

```
TestBed.configureTestingModule({
  imports: [ReactiveFormsModule],
  declarations: [SearchComponent]
});
```

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.searchText.value).toBe('Angular');
});
```

We use the `querySelector` method of the `nativeElement` property to find the `<input>` 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>` 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', () => {
  component.searchText.setValue('');
  expect(button.disabled).toBeTrue();
});

it('should log to the console', () => {
  const spy = spyOn(console, 'log');
  component.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, the nature of reactive forms makes them very easy to test since the form model is the single source of truth.

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.

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!

Join our community on Discord

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

`https://packt.link/LearningAngular4e`

13

Bringing an Application 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. But what do we mean by production build?

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 to a single bundle file.

In the previous chapters, we went through 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 code samples described in this chapter can be found in the `ch13` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-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. How does the compiler know which files to collect for the build process? It reads the `files` property of the `tsconfig.app.json` file, which indicates the main entry point of the Angular application:

```
/* To learn more about this file see: https://angular.io/config/tsconfig.
*/
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "outDir": "./out-tsc/app",
    "types": []
  },
  "files": [
    "src/main.ts"
  ],
  "include": [
    "src/**/*.d.ts"
  ]
}
```

From that point, it can 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:

```
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files           | Names         |  Raw Size | Estimated Transfer Size
main.7763e222c7fcca41.js      | main          | 116.53 kB |                34.94 kB
polyfills.df72b377d31a6849.js | polyfills     |  33.07 kB |                10.67 kB
runtime.771c5fa6dd11d129.js   | runtime       | 890 bytes |               510 bytes
styles.ef46db3751d8e999.css   | styles        |   0 bytes |                       -

                              | Initial Total | 150.46 kB |               46.11 kB
```

Figure 13.1: Build output

The preceding image 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
- `runtime`: Angular code that is needed for our application to run
- `styles`: Global CSS styles of our application

The Angular compiler outputs the resulting JavaScript files into a `dist\appName` folder, where `appName` is the application name. In addition to the **Initial Chunk Files** from the preceding image, it contains the following files:

- `3rdpartylicenses.txt`: A text file that contains software licenses for any libraries used in the application, including the Angular framework
- `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:

```
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files | Names         |  Raw Size
vendor.js           | vendor        |   1.59 MB |
polyfills.js        | polyfills     | 106.31 kB |
main.js             | main          |  47.30 kB |
runtime.js          | runtime       |   5.89 kB |
styles.css          | styles        | 778 bytes |

                    | Initial Total |   1.75 MB
```

Figure 13.2: Development build output

In the preceding image, 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 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`.

The first thing we need to do when working with environments is to create a `src\environments` folder in the Angular project and add an `environment.ts` file. The `environment.ts` file is the default environment of the application, which is used during development.

We can create additional environment files named `environment.env.ts` by convention, where env is a distinct name for the environment we want to add. For the staging environment, the filename would be `environment.staging.ts`.

Each environment file exports an `environment` object:

```
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 our Angular application. To use the `apiUrl` property in the main application component, we must do the following:

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

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

After creating the environment file, we must define the appropriate configuration in the `angular.json` file. It contains an `architect` property that defines basic CLI commands such as `serve`, `build`, and `test`. Each command may contain a configuration for each environment in a `configurations` property.

To add a new configuration for the staging environment, we add the following in the `configurations` property of the `build` command:

```
"staging": {
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.staging.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 `staging` environment. If we run the `ng build --configuration=staging` command, the Angular CLI will replace the `environment.ts` file with the `environment.staging.ts` file in the application bundle.

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 the preceding 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",
  "main": "src/main.ts",
  "polyfills": [
    "zone.js"
  ],
  "tsConfig": "tsconfig.app.json",
```

```
    "assets": [
      "src/favicon.ico",
      "src/assets"
    ],
    "styles": [
      "src/styles.css"
    ],
    "scripts": []
  }
```

The `options` object contains the following properties that we can use:

- `assets`: Contains static files of an Angular application such as icons, fonts, and translations. The Angular CLI includes the `favicon.ico` file and the `assets` folder by default.
- `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 Angular CLI **budgets**.

Budgets are thresholds that we can define in the `angular.json` configuration file and 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": "2kb",
    "maximumError": "4kb"
  }
]
```

The Angular CLI does a pretty good job of defining default budgets for us 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 `2` KB and an error when it goes over `4`.

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.io/guide/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.

As we can see, the Angular CLI does a tremendous job for us regarding build optimization. However, suppose the size of the final bundle remains considerably large. In that case, we can use the lazy-load module technique we have already seen in *Chapter 9*, *Navigate through Application with Routing*.

In a nutshell, we can use the Angular router to load Angular modules upon request when we are sure they will not be used at application startup. Thus, we dramatically reduce the initial bundle size because the Angular CLI creates one small bundle for each lazy-loaded module when building the application. For example, if we build the Angular application in *Chapter 9*, *Navigate through Application with Routing*, with the `ng build` command, the output will look like this:

```
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files           | Names             |  Raw Size | Estimated Transfer Size
main.ef4db3ac90182ddb.js      | main              | 244.39 kB |                66.68 kB
polyfills.df72b377d31a6849.js | polyfills         |  33.07 kB |                10.67 kB
runtime.fc13a61d6895c7a5.js   | runtime           |   2.62 kB |                 1.24 kB
styles.243c9b0382539700.css   | styles            | 656 bytes |               265 bytes

                              | Initial Total     | 280.72 kB |                78.85 kB

Lazy Chunk Files              | Names             |  Raw Size | Estimated Transfer Size
5.938a38f8c2d594b0.js         | about-about-module | 740 bytes |               344 bytes
```

Figure 13.3 : Build output with lazy-loaded module

In the preceding image, we can see that the Angular CLI has created a lazy chunk file named **about-about-module**, which is the bundle of the **about** module that is lazy-loaded by the router. If we had defined the module to be eagerly loaded, the lazy chunk file would not have been created, and it would have been included in the main bundle.

A good practice when we design an Angular application is to keep the bundle size as small as possible and plan accordingly. Consider which of the modules are not going to be used during startup and make them lazy-loaded. A good case for this is the menu links of a website. You can define one module for each link and load it lazily. As soon as you progress, if a module finally needs to be immediately available, make it eagerly loaded. In this way, you will always start with the smallest bundle size available.

The lazy-load technique also improves the launch time of an Angular application because a smaller bundle can be parsed faster from a browser.

When we have applied all previous optimizations but the final bundle remains large, the last-resort technique is using an external tool called **source-map-explorer**. It 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. Run the `source-map-explorer` binary against the main bundle file:

```
node_modules/.bin/source-map-explorer dist/my-app/main.*.js
```

It will open up a visual representation of the application bundle in the browser:

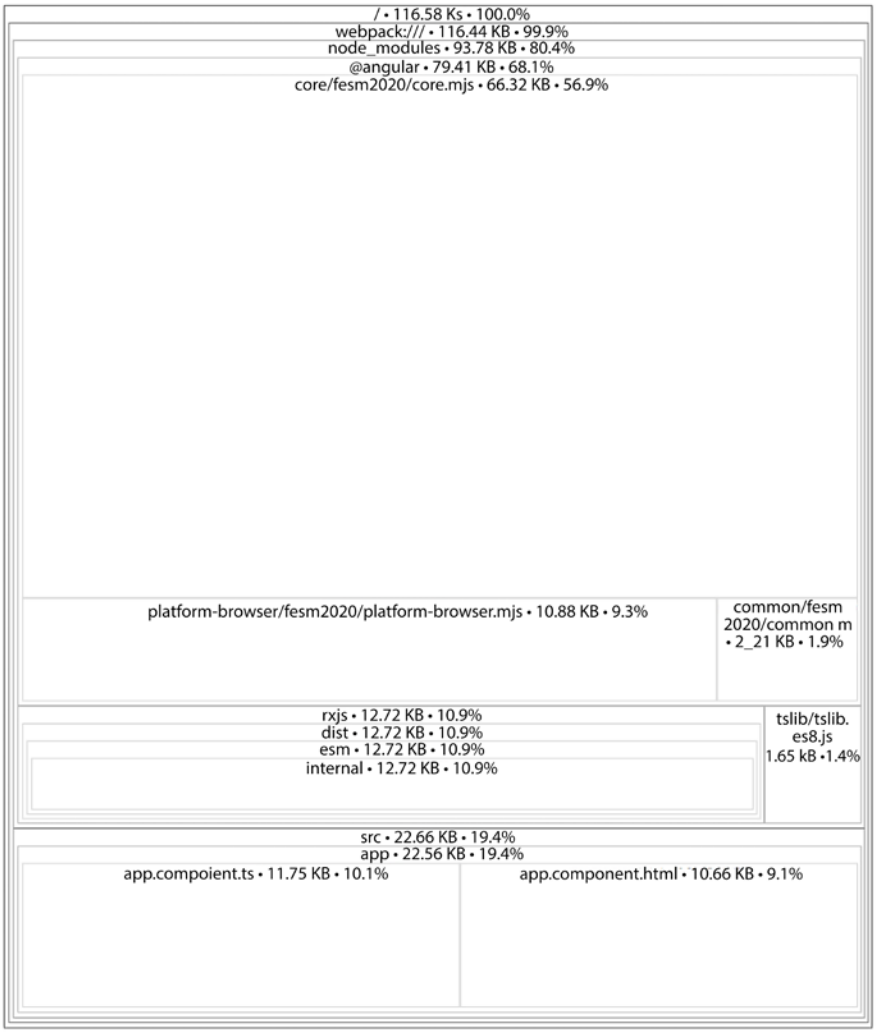


Figure 13.4: Source map explorer output

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-shaked 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` file:

```
"options": {
  "outputPath": "dist/my-app",
  "index": "src/index.html",
  "main": "src/main.ts",
  "baseHref": "/mypath/",
  "polyfills": [
    "zone.js"
  ],
  "tsConfig": "tsconfig.app.json",
  "assets": [
    "src/favicon.ico",
    "src/assets"
  ],
  "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 one of the following supported hosting providers:

- **Firebase**: `https://firebase.google.com/docs/hosting`
- **Vercel**: `https://vercel.com/solutions/angular`

- **Netlify**: `https://www.netlify.com`
- **GitHub**: `https://pages.github.com`
- **npm**: `https://npmjs.com`
- **Amazon**: `https://aws.amazon.com/s3/?nc2=h_ql_prod_st_s3`

The Angular community maintains npm packages for each of the previous hosting providers. You can find more details on how to install and use them at `https://angular.io/guide/deployment#automatic-deployment-with-the-cli`.

Summary

We finally took the last step toward completing our magical journey in the Angular framework. The deployment of an Angular application is the simplest and the most crucial part of the whole journey 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 handle application errors and debug Angular applications.

14

Handling Errors and Application Debugging

Application errors are an integral part of the lifetime of a web application. They can occur either during runtime or while developing the application. Runtime errors may happen due to an HTTP request that failed or an incomplete HTML form. Development errors usually happen when we do not properly use a programming language or framework according to its semantics. In both cases, we need a mechanism for debugging the application to investigate and fix the error.

In this chapter, we will learn how to handle different types of errors in an Angular application and understand errors that come from the framework itself. We will learn how to debug and profile an Angular application using Angular DevTools. We will explore the following concepts in more detail:

- Handling application errors
- Demystifying framework errors
- Debugging Angular applications

Technical requirements

The code samples described in this chapter can be found in the `ch14` folder of the following GitHub repository:

`https://www.github.com/PacktPublishing/Learning-Angular-Fourth-Edition`

Handling application errors

The most usual 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. In conjunction with the `pipe` operator, it can catch potential errors that may occur when initiating an HTTP request:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    })),
    catchError((error: HttpErrorResponse) => {
      console.error(error);
      return throwError(() => error);
    })
  );
}
```

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 rethrows the error as an observable. In this way, we ensure that the execution will continue and complete gracefully 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. The helper method would be used in every HTTP call of a service:

```
private handleError(error: HttpErrorResponse) {
  switch(error.status) {
    case HttpStatusCode.InternalServerError:
      console.error('Server error:', error.error);
      break;
    case HttpStatusCode.BadRequest:
      console.error('Request error:', error.error);
      break;
    default:
      console.error('Unknown error:', error.error);
  }
  return throwError(() => error);
}
```

The preceding method logs a different message in the browser console according to the `status` of the error. 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. The `HttpErrorResponse` and `HttpStatusCode` artifacts can be imported from the `@angular/common/http` namespace. The `HttpStatusCode` is an enumeration that contains a list of all HTTP status codes.

The `handleError` method currently manages HTTP errors that originate 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 modify the `handleError` method as follows:

```
private handleError(error: HttpErrorResponse) {
  switch(error.status) {
    case 0:
      console.error('Client error:', error.error);
      break;
    case HttpStatusCode.InternalServerError:
      console.error('Server error:', error.error);
      break;
    case HttpStatusCode.BadRequest:
      console.error('Request error:', error.error);
      break;
```

```
    default:
      console.error('Unknown error:', error.error);
  }
  return throwError(() => error);
}
```

In the preceding snippet, an error with status `0` indicates that it is an error that occurred in the client.

We can then add the `handleError` method to an HTTP call as follows:

```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    })),
    catchError(this.handleError)
  );
}
```

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(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
      return this.convertToProduct(product);
    })),
    retry(2),
    catchError(this.handleError)
  );
}
```

The point is that with the `catchError` operator, we can capture the error; how 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 like to 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 following 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 an Angular application.
2. 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 need to provide it later in the main application module.

4. Implement the `handleError` method from the `ErrorHandler` interface as follows:

```
handleError(error: any): void {
  const err = error.rejection || error;

  if (err instanceof HttpErrorResponse) {
    switch(err.status) {
      case 0:
        console.error('Client error:', error.error);
        break;
      case HttpStatusCode.InternalServerError:
        console.error('Server error:', error.error);
        break;
      case HttpStatusCode.BadRequest:
        console.error('Request error:', error.error);
        break;
      default:
```

```
        console.error('Unknown error:', error.error);
      }
    } else {
      console.error('Application error:', err)
    }
  }
```

5. In the preceding method, we check if the `error` object contains a `rejection` property. Errors that originate from the **Zone.js** library, which is responsible for the change detection in Angular, encapsulate the actual error inside that property.
6. After extracting the error in the `err` variable, we check to see if it is an HTTP error using the `HttpErrorResponse` type. Any errors thrown from HTTP calls using the `throwError` RxJS operator will eventually be caught in this check. All other errors are treated as application errors that occur client side.
7. Open the `app.module.ts` file and import the `ErrorHandler` artifact from the `@angular/core` npm package:

```
import { NgModule, ErrorHandler } from '@angular/core';
```

8. Import the custom error handler we created in the `app-error-handler.ts` file:

```
import { AppErrorHandler } from './app-error-handler';
```

9. Register the `AppErrorHandler` class as the global error handler of the application by adding it to the `providers` array of the `@NgModule` decorator:

```
providers: [
  { provide: ErrorHandler, useClass: AppErrorHandler }
]
```

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 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. The `AuthInterceptor` class in the `auth.interceptor.ts` file should be modified as follows:

```
export class AuthInterceptor implements HttpInterceptor {

  constructor(private authService: AuthService) {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler):
Observable<HttpEvent<unknown>> {
    const authReq = request.clone({
      setHeaders: { Authorization: 'myAuthToken' }
    });
    return next.handle(authReq).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === HttpStatusCode.Unauthorized) {
          this.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 from the `handle` observable. In all other errors, it will use the `throwError` operator to bubble up the error to the global error handler. 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 that 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 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 our applications. Sometimes things will go well, but other times they may cause errors in our application.

The Angular framework provides a mechanism for reporting some of these common errors with the following format:

NGWXYZ: {Error message}.<Link>

The preceding error format is analyzed as follows:

- **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 that is used for indexing 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 } from '@angular/core';
```

2. Implement the `ngAfterViewInit` method in the `AppComponent` class and change the `title` property inside the method body:

```
export class AppComponent implements AfterViewInit {
  title = 'my-app';

  ngAfterViewInit(): void {
    this.title = 'Learning Angular';
  }
}
```

3. Open the `app.component.html` file and add an `<h1>` element to display the value of the `title` property:

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

4. Run the `ng serve` command and navigate at `http://localhost:4200` using your browser.
5. Initially, everything looks to work correctly. The value of the title is displayed on the page correctly. Open the browser developer tools and select the **Console** tab. You should see the following output:

Figure 14.1: Console output

6. If we investigate the message, we will notice that changing the value of the `title` property from **my-app** to **Learning Angular** caused the error. Angular also points us to the specific place of the HTML template that caused the issue, as we can see from the line **at AppComponent_Template (app.component.html:1:5)**.

7. Clicking on the `https://angular.io/errors/NG0100` link will redirect us to the appropriate error guide in the Angular documentation:

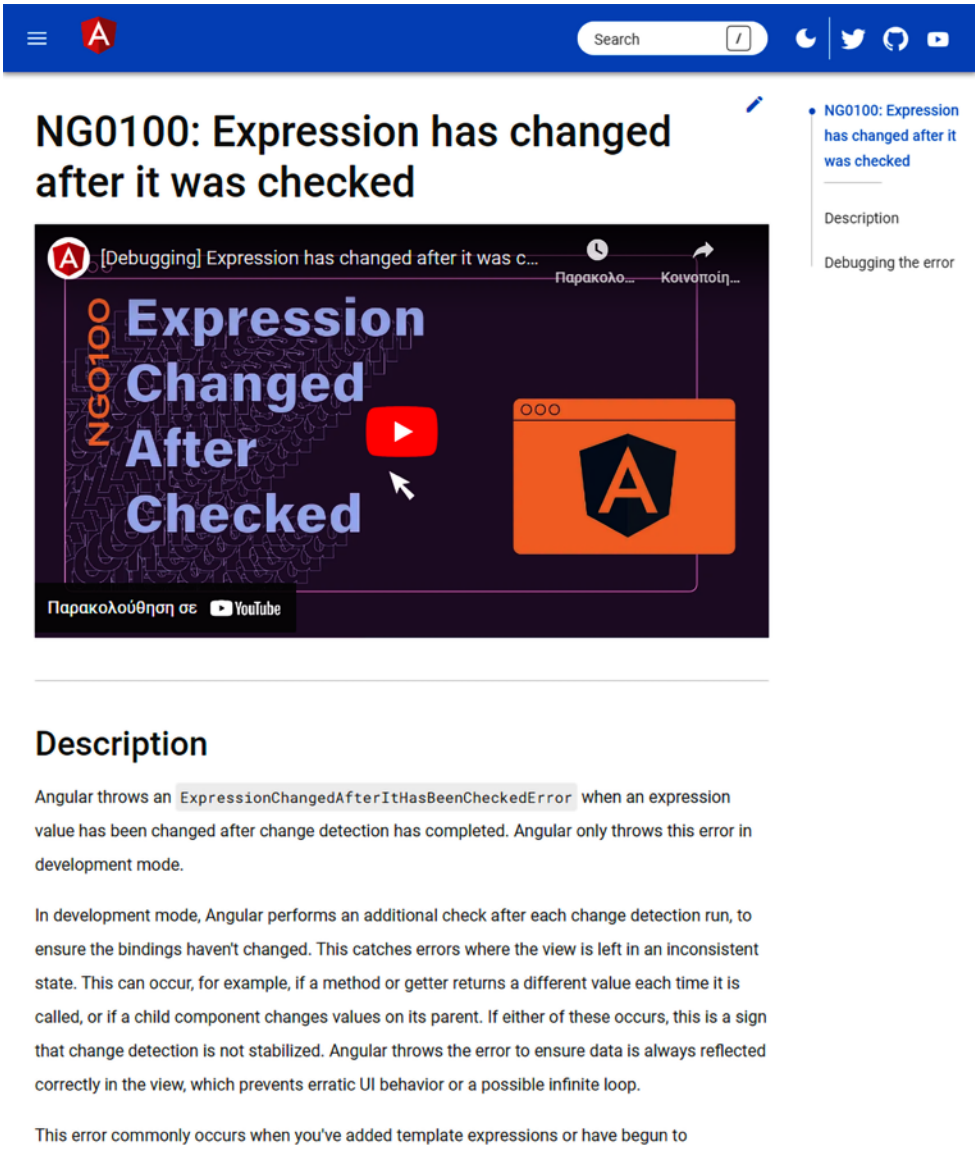


Figure 14.2: Error guide

The preceding error guide contains a detailed explanation of the specific error. It also describes how to fix the problem in our application code.

Sometimes, errors in an Angular application are difficult to spot and fix. We must debug the application to find and fix them in those cases. We will learn how to debug an Angular application in the following section.

Debugging Angular applications

We can debug an Angular application using standard debugging techniques for web applications or the tooling that the Angular framework provides out of the box. Both approaches should work the same, and the choice depends on the use case. In this section, we will learn about the following debugging methods:

- Use the `console` object to print data and messages to the browser console.
- Use the browser developer tools to add breakpoints in the application source code and inspect it.
- Use the Angular DevTools extension for debugging and profiling.

We will start by learning how to use the `console` object for debugging purposes.

Using the Console API

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.

We have already used the `console` object in several chapters of this book for demonstration purposes. However, it is considered an old-fashioned approach, and a codebase with many `console.log` methods is not readable. An alternate way is to use breakpoints in our source code, as we will learn in the following section.

Adding breakpoints in source code

We can add breakpoints inside the source code from the browser developer tools 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. We will see how to add breakpoints in **Google Chrome** using an example:

1. Run the `ng serve` command to start the Angular application.

2. Open Google Chrome and navigate to `http://localhost:4200`.
3. Press the *F12* keyboard key to open the developer tools.
4. Select the **Sources** tab and then the **webpack://** option from the **Page** tab.
5. Open the `app.component.ts` file that exists inside the `src\app` folder.
6. Left-click with your mouse on line **12** to add a breakpoint:

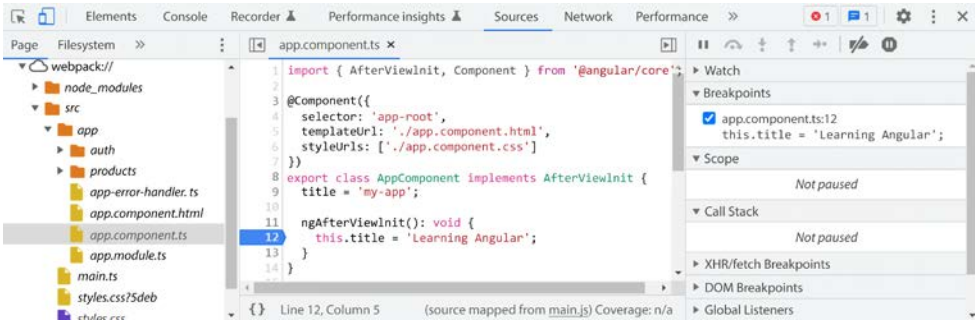


Figure 14.3 : Sources tab

7. Refresh the browser, and you will notice that the Chrome debugger hits the breakpoint and pauses the application execution:

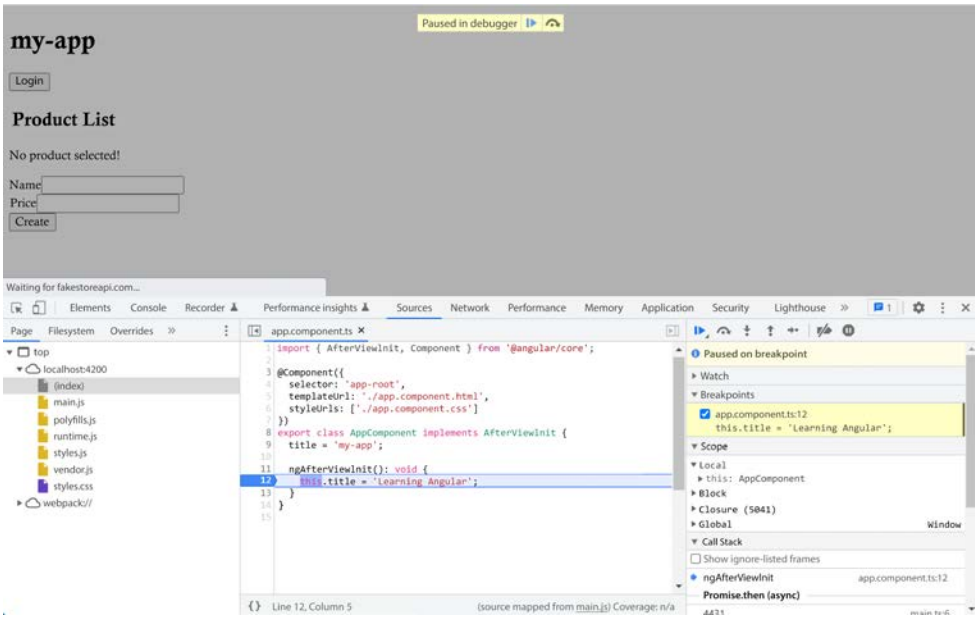


Figure 14.4: Chrome debugger

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

The Angular team has created a tool that complements the debug process with breakpoints, as we will learn in the following section.

Using Angular DevTools

Angular DevTools is a browser extension created and maintained by the Angular team. It allows us to debug 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/ienfalfjdbdpebioblfackkekamfmbnh`
- **Mozilla Firefox**: `https://addons.mozilla.org/en-GB/firefox/addon/angular-devtools`

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

- **Components**: Displays the component tree of the Angular application
- **Profiler**: Allows us to profile and inspect the Angular application

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 on the right-hand side:

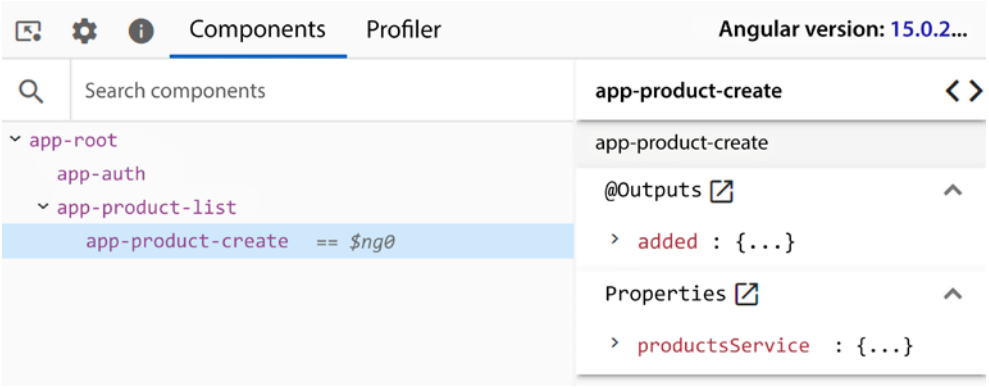


Figure 14.5: Component preview

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 in the **Sources** tab:

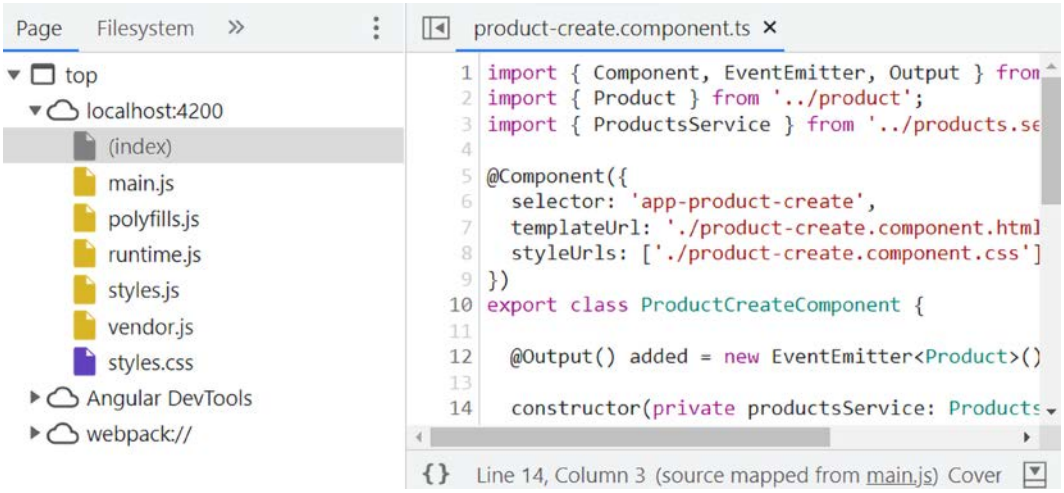


Figure 14.6: Sources tab

We can then add breakpoints in the source code and debug our application.

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

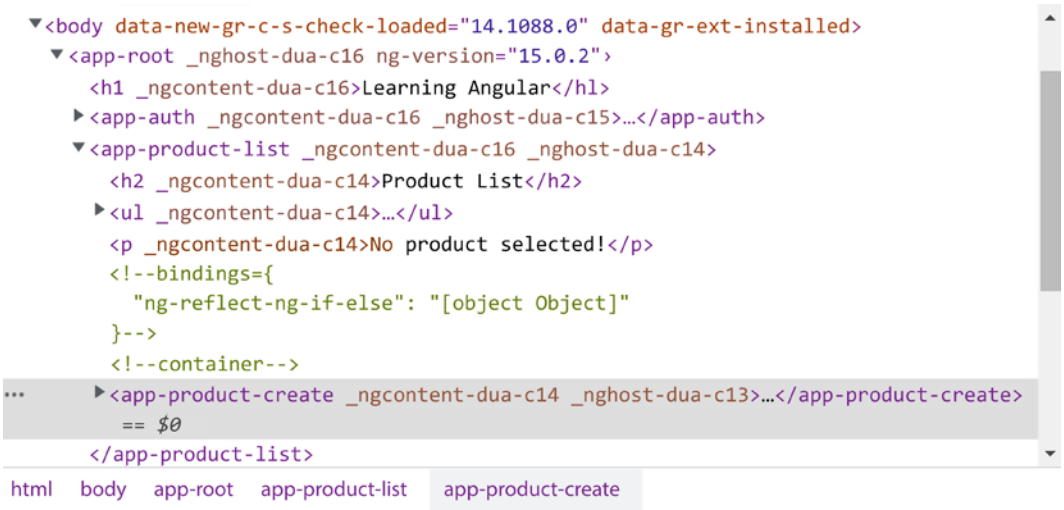


Figure 14.7: Elements tab

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:

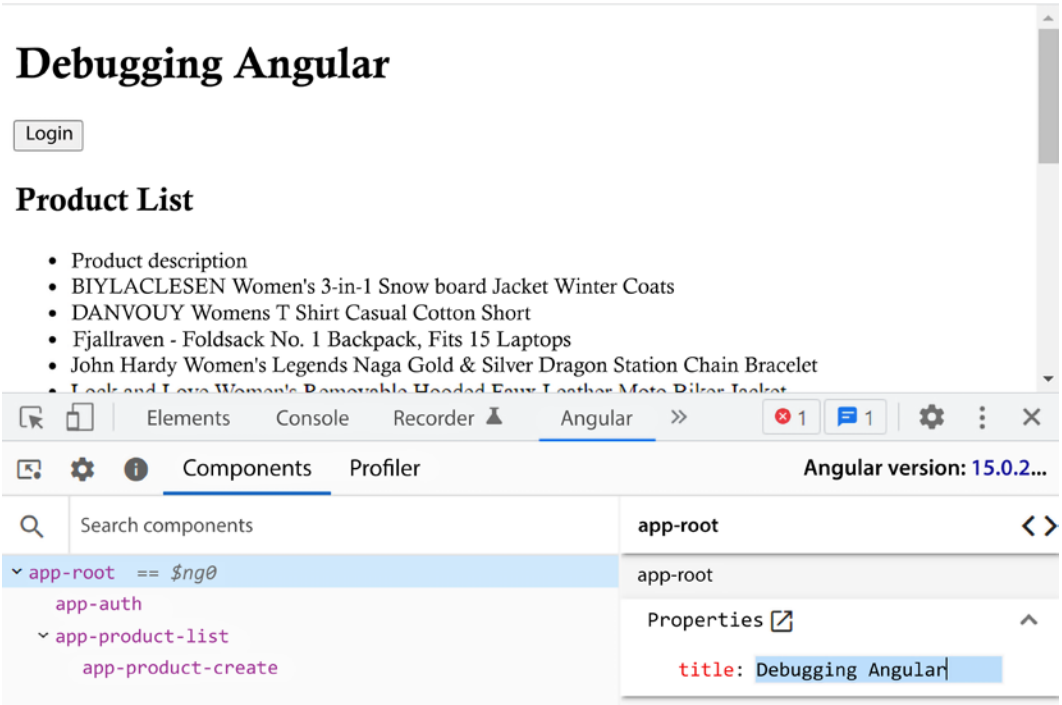


Figure 14.8: Change component state

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

Another useful feature of the Angular DevTools is the Profiler. The profiler allows us to profile and inspect an Angular application in terms of performance. We can use it to examine how an application behaves during the change detection mechanism and check for any bottlenecks.

We can access the Angular DevTools profiler from the **Profiler** tab. To start profiling an Angular application, we must do the following:

1. Click the **Record** button represented by the circle dot.

   Start using the Angular application normally as a user and focus on those parts that seem to have performance issues.

2. Click the **Record** button once again to stop the profiler recording.

When the profiler stops recording, it creates a graph of bars representing the change detection cycles of the application. The height of each bar indicates the amount of time spent in the cycle. Selecting a bar will display the components and directives involved during that cycle:

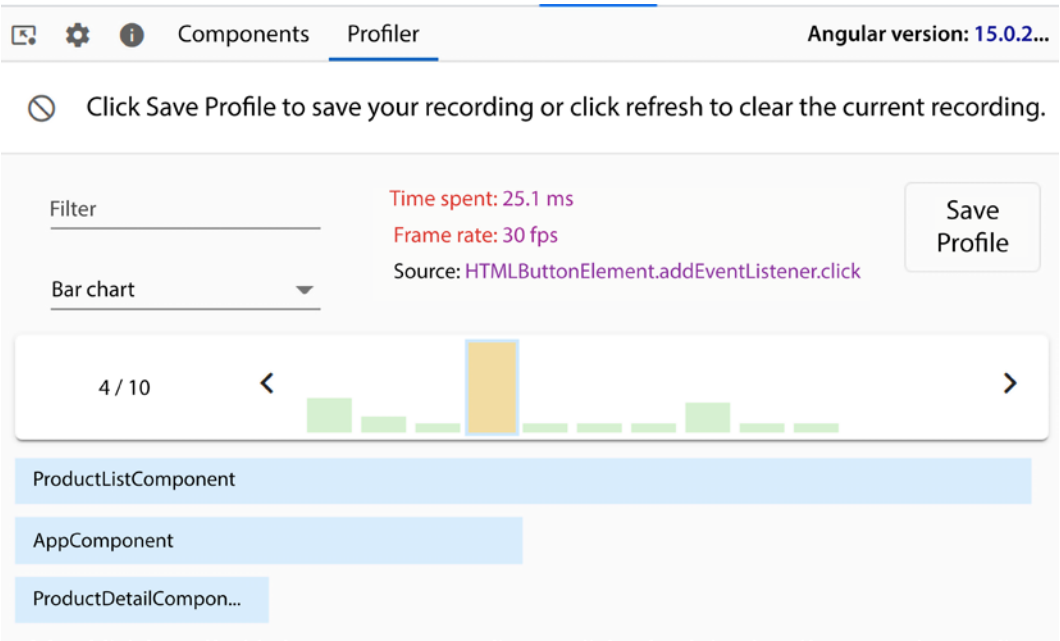


Figure 14.9: Change detection cycle graph

In the preceding image, we see that it also displays statistics about the selected change detection cycle, such as the time spent and the source that triggered the cycle.

If we click on a specific component, we will see a detailed view, including how much time was spent in the change detection cycle and its parent components in the tree hierarchy:

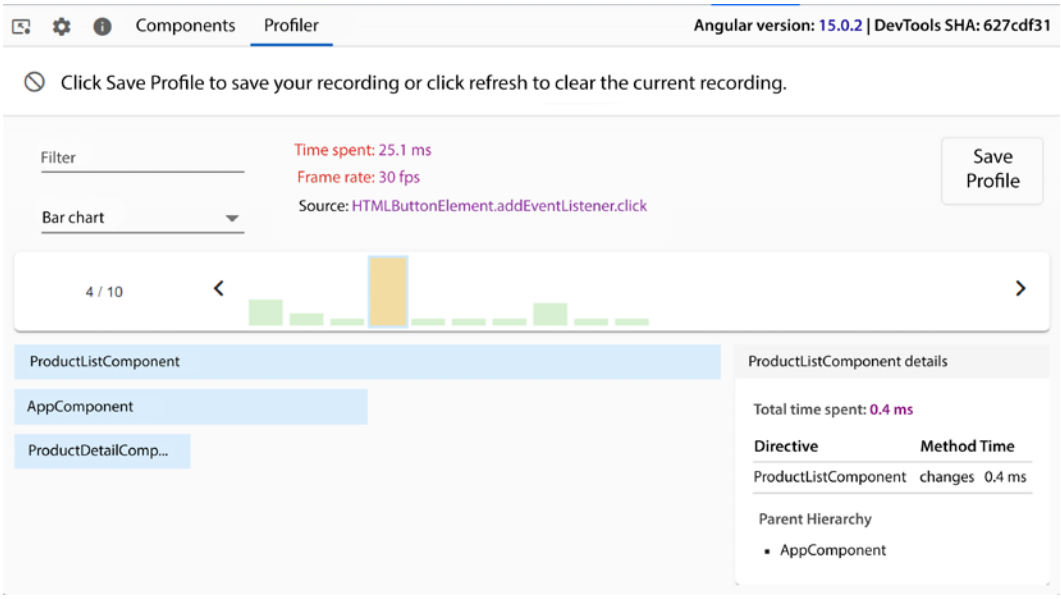


Figure 14.10: Component details

Angular DevTools, along with the Angular CLI, is a tool that should not be overlooked by any developer that works with the Angular framework. The Angular CLI allows the developer to scaffold, test, and build an Angular application. Angular DevTools provides added value in this workflow by allowing developers to detect and investigate development issues.

Summary

Handling errors during runtime or development is crucial for every Angular application. The knowledge of how to debug an application for detecting issues and problems is essential for every Angular developer.

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. Finally, we explored different ways of debugging an Angular application, from the Console API to the more sophisticated Angular DevTools.

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 their toolchains. We were delighted to have you on board, and we hope this book has broadened your idea of what an excellent framework such as Angular can offer!

Join our community on Discord

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

`https://packt.link/LearningAngular4e`

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

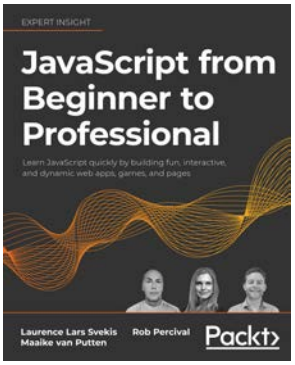
If you enjoyed this book, you may be interested in these other books by Packt:

Responsive Web Design with HTML5 and CSS, Fourth Edition

Benjamin Frain

ISBN: 9781803242712

- Use media queries, including detection for touch/mouse and color preference
- Learn HTML semantics and author accessible markup
- Facilitate different images depending on screen size or resolution
- Write the latest color functions, mix colors, and choose the most accessible ones
- Use SVGs in designs to provide resolution-independent images
- Create and use CSS custom properties, making use of new CSS functions including ‘clamp’, ‘min’, and ‘max’
- Add validation and interface elements to HTML forms
- Enhance interface elements with filters, shadows, and animations



JavaScript from Beginner to Professional

Laurence Svekis, Maaike Putten, Rob Percival

ISBN: 9781800562523

- Use logic statements to make decisions within your code
- Save time with JavaScript loops by avoiding writing the same code repeatedly
- Use JavaScript functions and methods to selectively execute code
- Connect to HTML5 elements and bring your own web pages to life with interactive content
- Make your search patterns more effective with regular expressions
- Explore concurrency and asynchronous programming to process events efficiently and improve performance
- Get a head start on your next steps with primers on key libraries, frameworks, and APIs

Django 4 By Example, Fourth Edition

Antonio Melé

ISBN: 9781801813051

- Learn Django essentials, including models, ORM, views, templates, URLs, forms, authentication, signals and middleware
- Implement different modules of the Django framework to solve specific problems
- Integrate third-party Django applications into your project
- Build asynchronous (ASGI) applications with Django
- Set up a production environment for your projects
- Easily create complex web applications to solve real use cases

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, Fourth 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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

S

T

U

V

W

Z

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/9781803240602`

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