Learning Angular 2

Your quick, no-nonsense guide to building real-world apps with Angular 2

Pablo Deeleman

Learning Angular 2

Table of Contents

Learning Angular 2

Learning Angular 2

Credits

Author

Pablo Deeleman

Reviewer

Johannes Weber

Commissioning Editor

Sarah Crofton

Acquisition Editor

Reshma Raman

Content Development Editor

Samantha Gonsalves

Technical Editor

Mohita Vyas

Copy Editors

Roshni Banerjee

Akshata Lobo

Project Coordinator

Sanchita Mandal

Proofreader

Safis Editing

Indexer

Priya Sane

Graphics

Kirk D'Penha

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Pablo Deeleman is a former UX designer and frontend engineer who discovered the Web back in the 90s, when a 14,400 bps modem was the key to an unparalleled world of marvels and a build-your-own-website was the name of the game.

After getting his BA (Hons) degree in marketing and moving through different roles in the advertising arena, he took his chance and evolved into a self-taught, passionate UX designer and frontend developer with a crunch for beautifully crafted CSS layouts and JavaScript thick clients, having produced countless interactive designs and web desktop and mobile applications ever since.

During these years, he has fulfilled his career as both an UX designer and frontend developer by successfully leading Internet projects for a wide range of clients and teams, encompassing European online travel operators, Silicon Valley-based start-ups, international heavy-traffic tube websites, global banking portals, or gambling and mobile gaming companies, just to name a few. At some point along this journey, the rise of Node.js and single-page-application frameworks became a turning point in his career, being currently focused on building JavaScript-driven web experiences.

After having lived and worked in several countries, Pablo Deeleman currently lives in Barcelona, Spain, where he leads the frontend endeavor in the Barcelona studio of Gameloft, the world leader in mobile gaming, and the home of internationally acclaimed games, such as Despicable Me: Minions Rush and Asphalt 8.

When not writing books or taking part in industry events and talks, he spends most of his time fulfilling his other passion: playing piano and guitar.

Acknowledgments

About the Reviewer

Johannes Weber is a passionate developer and adviser in the field of web technologies spotlighted on enterprise JS apps. He works for Mayflower GmbH (Munich, Germany), where he focuses on the migration of SPA and MPA. In his free time, he (co)organizes the AngularJS Munich meetups, AngularCamp and JS-Kongress.de. Johannes cofounded ESnextNews.com, where you get five great ECMAScript.next links every week in your inbox.

eBooks, discount offers, and more

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

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

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

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

This book is dedicated to my parents Paul and Pepa, and in the loving memory of my brother José Raúl.

You will live forever in our hearts.

Preface

Over the past years, Angular 1.x has became one of the most ubiquitous JavaScript frameworks for building cutting edge web applications, either big or small. At some point, its shortcomings with regard to performance and scalability became too prominent as soon as applications grew in size and complexity. Angular 2 was then conceived as a full rewrite from scratch to fulfill the expectations of modern developers, who demand blazing fast performance and responsiveness in their web applications.

Angular 2 has been designed with modern web standards in mind and allows full flexibility when picking up your language of choice, providing full support for ES6 and TypeScript, but working equally well with today's ES5, Dart, or CoffeeScript. Its built-in dependency injection functionalities let the user build highly scalable and modular applications with an expressive and self-explanatory code, turning maintainability tasks into a breeze, while simplifying test-driven development to the max. However, where Angular 2 stands out is when it shows off its unparalleled level of speed and performance, thanks to its new change detection system that is up to five times faster than its previous incarnation. Cleaner views and an unsurpassed standards-compliant templating syntax compound an endless list of powerful features for building the next generation of web mobile and desktop apps.

Angular 2 is here to stay and will become a game changer in the way modern web applications are envisioned and developed in the years to come. However, and due to its disruptive design and architecture, learning Angular 2 might seem a daunting effort to newcomers.

This is where this book comes in—its goal is to avoid bloating the reader with API references and framework descriptions, but to embrace a hands-on approach, helping the reader learn how to leverage the framework to build stuff that matters right from day one. This is learning by doing right from the start.

This book aims to give developers a complete walkthrough of this new platform and its TypeScript-flavored syntax by building a web project from back to forth, starting from the basic concepts and sample components and iterating on them to build up more complex functionalities in every chapter until we launch a complete, tested, production-ready sample web application by the end of the book.

What this book covers

Chapter 1, *Creating Our Very First Component in Angular 2,* introduces the reader to web components, which are the building blocks of all Angular 2 applications.

Chapter 2, *Introducing TypeScript,* instructs the reader about the syntax and particularities of this typed superset of ECMAScript 6, being in fact the syntax of choice of the Angular team for building Angular 2.

Chapter 3, *Implementing Properties and Events in Our Components,* describes how our components behave like state machines that can change their state by receiving data through their input properties and emit data as events through their output properties.

Chapter 4, *Enhancing our Components with Pipes and Directives,* gives a complete walkthrough of the framework's built-in pipes used to digest data output in our templates, and also the built-in directives that provide advanced functionality to our component. The reader will also learn how to create custom pipes or directives

Chapter 5, *Building an Application with Angular 2 Components,* devotes an entire chapter to recap what we have learned so far and orchestrates everything to ensure our Angular 2 projects scale well regardless their size and conform to the community coding and naming conventions.

Chapter 6, *Asynchronous Data Services with Angular 2,* teaches the reader how to implement and deploy HTTP connections with other data services by means of the Http module, so we can create our own data service clients.

Chapter 7, *Routing in Angular 2,* introduces the reader to Angular 2's router and its built-in directives, providing a complete walkthrough the different strategies we have to load components from routes and handle the state through the History API.

Chapter 8, *Forms and Authentication Handling in Angular 2,* illustrates the different strategies we have at our disposal to build web forms with Angular 2, manage two-way data binding on input controls, and create complex forms and validations.

Chapter 9, *Animating Components with Angular 2,* covers the currently available tools and classes for implementing animations on our components, from pure CSS animations handled with Angular 2 directives to more complex transitions purely managed through JavaScript, thanks to Angular 2 animation builders.

Chapter 10, *Unit Testing in Angular 2,* will guide the reader through the steps required for implementing a sound testing foundation in our application, and the general patterns for deploying unit tests on components, directives, pipes, routes, and services.

What you need for this book

In order to develop the examples contained in this book, you will primarily need a web browser updated to its latest version. We recommend Google Chrome or Mozilla Firefox, although Angular 2 is meant to be supported in all evergreen browsers.

You will also need terminal software installed on your OS, since many operations are handled through npm commands to the console. In this sense, having Node.js and npm installed on your system will be required to run most of the console commands mentioned in the book. The rest of modules required and their installation procedure will be described as we go.

Last, but not least, you will require a text editor to code your Angular 2 modules, although Chapter 1, *Creating Our Very First Component in Angular 2* will provide a thorough walkthrough of all the best IDE alternatives in store nowadays for developing Angular 2 applications.

Who this book is for

This book is targeted at web developers who want to build the next generation of state-of-the-art mobile and desktop web applications with Angular 2. This book does not require you to have prior exposure to either Angular 1.x or 2, although comprehensive knowledge of JavaScript is assumed. It's great for newcomers to Angular who learn best through clear explanations and definition of concepts.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "As a result of this action, we will find a new `tsconfig.json` file at the root of our project, including the settings required by the TypeScript compiler to transpile the component code into plain ECMAScript 5 JavaScript code readable by current browsers."

A block of code is set as follows:

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

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

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

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

```
$ npm install angular2 es6-shim es6-promise reflect-metadata rxjs zone.js --save
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The **learn** section gives us access to a quick tutorial to get up to speed with the language in no time."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

You can download the example code files for this book from GitHub at
https://github.com/deeleman/ learning-angular2.

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

You can download the code files by following these steps:

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

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

Errata

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

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

Piracy

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

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

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

Questions

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

Chapter 1. Creating Our Very First Component in Angular 2

Unless you were lost in space for the past couple of years, chances are you are well aware of the momentum that modern JavaScript web frameworks and libraries have got in the frontend arena nowadays. We have even reached a stage where a new framework is born every day, forcing frontend developers to assess carefully whether this new cutting-edge code toolkit adds enough value to justify the time and effort required to face its learning curve and put it to good use in our next project.

Eventually, a handful of names ended up gaining more relevance than the rest. We are obviously referring to client-side frameworks that will probably sound pretty familiar to you already: Backbone, Ember, Knockout, Angular 1, and so on.

As the battle for supremacy in the JavaScript world carried on, new frameworks such as React or Aurelia entered the game, favoring web components and harnessing the power of Shadow DOM as the cornerstone of its architecture. Applications built this way proved to be more modular, scalable, and maintainable, let alone their unparalleled level of performance.

Angular 1 had come a long way already since its inception and its shortcomings had become too prominent to be overlooked any longer. It was time for something better and a simple revamp of codebase did not suffice. A more ambitious approach was required and Angular 2 was developed—a new framework engineered from scratch, which fully embraces the newest trends in the industry. It has web components at the heart of its design and it harnesses the power of Shadow DOM to maximize the responsiveness of our web entities against state changes. On top of that, Angular 2 offers a state-of-the-art change detection system baked in to each component, which is responsible for propagating bindings throughout the tree of components that comprise our applications.

The defining traits of Angular 2 go beyond the concept of just being a mere web components framework, since its features encompass pretty much everything you need in a modern web application: component interoperability, universal support for multiple platforms and devices, a top-notch dependency injection machinery, a flexible but advanced router mechanism with support for decoupling and componentization of route definitions, advanced HTTP messaging, and animation or internationalization, just to name a few.

In this chapter, we will:

- Learn why Angular 2 is so unique in comparison to its previous versions
- Learn how to set up our code environment to work with Angular 2 and TypeScript
- Enhance our IDE of choice to provide a better experience coding Angular 2 apps
- Build our very first Angular 2 web component and learn how to embed it on a web page
- Add basic interactivity features to our web component
- Discover some basic helpers to better format the data output

A fresh start

As mentioned before, Angular 2 represents a full rewrite of the Angular 1.x framework, introducing a brand new application architecture completely built from scratch in TypeScript, a strict superset of JavaScript that adds optional static typing and support for interfaces and decorators.

In a nutshell, Angular 2 applications are based on an architecture design that comprises trees of web components interconnected between them by their own particular I/O interface. Each component takes advantage under the covers of a completely revamped dependency injection mechanism. To be fair, this is a simplistic description of what Angular 2 really is. However, the simplest project ever made in Angular is cut out by these definition traits. We will focus on learning how to build interoperable components and manage dependency injection in the next chapters, before moving on to routing, web forms, or HTTP communication. This also explains why we will not make explicit references to Angular 1.x throughout the book. Obviously, it makes no sense to waste time and pages referring to something that will not provide any useful insights on the topic, besides the fact we assume that you might not know about Angular 1.x, so such knowledge does not have any value here.

Web components

Web components is a concept that encompasses four technologies designed to be used together to build feature elements with a higher level of visual expressivity and reusability, thereby leading to a more modular, consistent, and maintainable web. These four technologies are as follows:

- **Templates**: These are pieces of HTML that structure the content we aim to render
- **Custom Elements**: These templates not only contain traditional HTML elements, but also the custom wrapper items that provide further presentation elements or API functionalities
- **Shadow DOM**: This provides a sandbox to encapsulate the CSS layout rules and JavaScript behaviors of each custom element
- **HTML Imports**: HTML is no longer constrained to host HTML elements, but to other HTML documents as well

In theory, an Angular 2 component is indeed a custom element that contains a template to host the HTML structure of its layout, the latter being governed by a scoped CSS style sheet encapsulated within a Shadow DOM container. Let's try to rephrase this in plain English. Think of the range input control type in HTML5. It is a handy way to give our users a convenient input control for entering a value ranging between two predefined boundaries. If you have not used it before, insert the following piece of markup in a blank HTML template and load it in your browser:

```
<input id="mySlider" type="range" min="0" max="100" step="10">
```

You will see a nice input control featuring a horizontal slider in your browser. Inspecting such control with the browser developer tools will unveil a concealed set of HTML tags that were not present at the time you edited your HTML template. There you have an example of Shadow DOM in action, with an actual HTML template governed by its own encapsulated CSS with advanced dragging functionality. You will probably agree that it would be cool to do that yourself. Well, good news is that Angular 2 gives you the toolset required for delivering this very same functionality, so we can build our own custom elements (input controls, personalized tags, and self-contained widgets) featuring the inner HTML markup of our choice and a very own stylesheet that does not affect (nor is impacted) by the CSS of the page hosting our component.

Why TypeScript over other syntaxes?

Angular 2 applications can be coded in a wide variety of languages and syntaxes: ECMAScript 5, Dart, ECMAScript 6, TypeScript, or ECMAScript 7.

TypeScript is a typed superset of ECMAScript 6 (also known as ECMAScript 2015) that compiles to plain JavaScript and is widely supported by modern OSes. It features a sound object-oriented design and supports annotations, decorators, and type checking.

The reason why we picked (and obviously recommend) TypeScript as the syntax of choice for instructing how to develop Angular 2 applications in this book is based on the fact that Angular 2 itself is written in this language. Being proficient in TypeScript will give the developer an enormous advantage when it comes to understanding the guts of the framework.

On the other hand, it is worth remarking that TypeScript's support for annotations and type introspection turns out to be paramount when it comes to managing dependency injection and type binding between components with a minimum code footprint, as we will see further down the line in this book.

Ultimately, you can carry out your Angular 2 projects in plain ECMAScript 6 syntax if that is your preference. Even the examples provided in the book can be easily ported to ES6 by removing type annotations and interfaces, or replacing the way dependency injection is handled in TypeScript with the most verbose ES6 way.

Note

For the sake of brevity, we will only cover examples written in TypeScript and actually recommend its use because of its higher expressivity thanks to type annotations, and its neat way of approaching dependency injection based on type introspection out of such type annotations.

Setting up our workspace

Before jumping into the implementation of our very first and shiny Angular 2 component, we need to bring in all the tools we will require to implement software based on TypeScript, let alone the Angular 2 framework modules themselves.

First and foremost, create a folder and double check that the NPM CLI is available in your system and is properly updated to the latest stable version. Otherwise, please go to https://nodejs.org and install the latest Node.js runtime.

Note

At the time of writing, the Angular 2 framework is in Release Candidate 1 version, so the requirements for building and deploying the examples contained in this book might have changed overnight. The author maintains a code repository at https://github.com/deeleman/learning-angular2, where you can check the most up-to-date version of each example contained in this book. The repository is divided into chapter folders and each folder contains the incremental version of the project as it is at the end of each chapter. Please refer to the code repository should any problem arise upon installing or deploying the examples in the book.

Installing dependencies

Our first requirement will obviously be to install Angular 2 onto our workspace, including its own peer dependencies. The Angular 2 team has made a great effort to ensure the installation is modular enough to allow us to bring only what we need, becoming our projects more or less lean depending on the requirements.

In this sense, Angular 2 does not come in the form of a single installable package, but many. This gives the smart developer the opportunity to pick only those modules that are required for its project, minifying the overall dependencies footprint. Some of these packages, such as `common` or `core`, are required regardless the type of project we want to ship. Some others, such as `platform-browser-dynamic`, are bound to the type of project and target platform addressed. A nonthorough list of the most common packages that you will require in your projects is given here:

- `@angular/core`: This is the most relevant package, encompassing the backbone of Angular and its most common elements, such as directives and components. You will need to rely on this module on a common basis to import the basic elements of Angular 2 into your project.
- `@angular/common`: You will seldom need to explicitly import tokens from this module, but it is worth remarking that this package contains the definitions of all the directives, services, and pipes contained by Angular 2, among other relevant classes.
- `@angular/compiler`: Same as `common`, you will rarely import tokens explicitly from this module, although it is the one responsible for compiling the HTML templates and turning them into code that can render the application's UI output.
- `@angular/platform-browser`: This module contains classes and functions required for composing and interacting with the DOM in a web browser context. Updating the page title or configuring the touch gestures setup are common actions made possible by this module. This package also contains the functions required to compile templates offline in production environments.
- `@angular/platform-browser-dynamic`: We will rely thoroughly on this module during the course of the book, since it will provide us with the bootstrapping function we will require to initialize our applications on development.
- `@angular/http`: It is the Angular 2 HTTP client, which we will cover in detail in Chapter 6, *Asynchronous Data Services with Angular 2*.
- `@angular/router`: It is the Angular 2 built-in router still under Beta at the time of this writing.
- `@angular/router-deprecated`: A snapshot of the previous incarnation of the Angular 2 built-in router, made available to ensure backward compatibility with existing applications. Chapter 7, *Routing in Angular 2*, will cover it in detail and explain some of its most remarkable differences with the new router still under development.

At the time of writing, these are all the different third-party libraries that are required as peer dependencies in an Angular 2 project, apart from the Angular 2 modules:

- `es6-shim`: This introduces ECMAScript 6 compatibility polyfills for legacy JavaScript engines (mostly Microsoft Internet Explorer). This dependency is now required because many major browsers still do not provide wide support for ECMAScript 6 features, but hopefully, this will change soon. Some other implementations use the core-js standard library instead. Ultimately, pick the one you like the most as long as it properly polyfills the core ES2015 APIs required by Angular 2.
- `zone.js`: This is a polyfill for the Zone specification that is used to handle change detection in Angular 2 applications.
- `reflect-metadata`: This brings support for decorators in our Angular 2 classes and metadata reflection in our components. We will see decorators in action later on in this chapter and a broader overview of its different types and implementations in Chapter 2, *Introducing TypeScript*. Decorators are a core part of Angular 2.
- `rxjs`: This library was developed by Microsoft Open Technologies, Inc. According to Microsoft, it is a set of libraries to compose asynchronous and event-based programs using observable collections and Array#extras style composition in JavaScript. In short, RxJS is a library for managing Observables, which allow us to make our applications fully reactive to asynchronous state changes. The Observables spec will be standardized by modern browsers in the future, so we will be able to rule out this dependency by then.

These dependencies may evolve without prior notice so please refer to the GitHub repository for the most up-to-date list of requirements.

Note

You will be probably surprised by the amount of libraries that Angular 2 does need and the fact that these dependencies are not part of the Angular bundle itself. This is because these requisites are not specific to Angular 2, but of a vast majority of modern JavaScript applications nowadays.

With all these dependencies and third-party libraries in mind, you can run the following set of bash commands in your terminal console, once you have created a folder for the project we will cover in this book:

```
$ mkdir learning-angular2
$ cd learning-angular2
$ npm init
$ npm install @angular/common @angular/core @angular/compiler --save
$ npm install @angular/platform-browser @angular/platform-browser-dynamic --save
$ npm install @angular/router @angular/router-deprecated --save
$ npm install @angular/http --save
$ npm install es6-shim reflect-metadata rxjs zone.js --save
```

Apart from the dependencies enlisted previously, we will also need to install the systemjs universal module loader package in order to support module loading between code units once transpiled into ES5. The systemjs package is not the only option available for managing module loading in Angular 2. In fact, we can swap it for other module loaders, such as

WebPack (https://webpack.github.io/), although all the examples provided in this book make use of SystemJS for handling code injection. We will install SystemJS, flagging it as a development dependency by executing the following command:

```
$ npm install systemjs –save
```

Last, but not least, we will also install **Bootstrap** in our application so that we can easily craft a nice UI for the example application we will build incrementally in each chapter. This is not an Angular 2 requirement, but a particular dependency of the project we will carry out throughout this book:

```
$ npm install bootstrap –save
```

The installation can throw different alerts and warnings depending on the versions of each peer dependency required by Angular 2 at this moment in time, so in case of issues, I strongly recommend to fetch the latest version of the package.json file available in this book's code repository https://github.com/deeleman/learning-angular2/blob/master/chapter_01/package.json.

Download the file to your directory workspace and run the npm install command. NPM will find and install all the dependencies for you automatically.

Note

Mac OS users, who have not claimed ownership rights on the npm directory located at /usr/local/bin/npm (or /usr/local/npm for those users on OS versions prior to Mac OS El Capitan), might need to execute the npm install command with sudo privileges.

Installing TypeScript

We have now a complete set of Angular 2 sources and their dependencies, plus the Bootstrap module to beautify our project and SystemJS to handle module loading and bundle generation.

However, TypeScript is probably not available on your system yet. Let's install TypeScript and make it globally available on your environment so that we can leverage its convenient CLI to compile our files later on:

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

Great! We're almost done. One last step entails informing TypeScript about how we want to use the compiler within our project. To do so, just execute the following one-time command:

```
$ tsc --init --experimentalDecorators --emitDecoratorMetadata --target ES5 --module system --moduleResolution node
```

Basically, we have just initialized a TypeScript project (which is our Angular 2 project itself) with support for experimental decorators (as we mentioned already, these are a new feature in ES7 and TypeScript that Angular 2 uses extensively) and set SystemJS as the default mechanism for importing modules and dependencies between files.

As a result of this action, we will find a new `tsconfig.json` file at the root of our project, including the settings required by the TypeScript compiler to transpile the component code into plain ECMAScript 5 JavaScript code readable by current browsers.

Note

Please remember that our browsers do not provide support for TypeScript or ECMAScript 6 out of the box, so we will transpile our code to some flavor of JavaScript that is widely supported by our target browsers.

A sneak peek on such file will yield the following:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "noImplicitAny": false,
    "outDir": "built",
    "rootDir": ".",
    "sourceMap": false
  },
  "exclude": [
    "node_modules"
  ]
```

```
}
```

Simple, right? The set of properties included in our config manifest is self-descriptive enough, but we can highlight three interesting properties. They are as follows:

- `rootDir`: This points to the folder the compiler will use to scan for TypeScript files to compile (currently the base folder in our example).
- `outDir`: This defines where the compiled files will be moved unless we define our own output path by means of the `--outDir` parameter in the command line, the compiler will default to the `built` folder created at runtime in the same location where the `tsconfig.json` file lives.
- `sourceMap`: This sets the source code mapping preferences to help debugging.

Toggle its value to `true` if you want source map files to be generated at runtime to back trace the code to its source through the browser's dev tools in case exceptions arise.

Besides these properties, we also can see that we have marked the `node_modules` folder as excluded, which means that the `tsc` command will skip that folder and all its contents when transpiling TypeScript files to ES5 throughout the application tree.

Tip

I would encourage you to refer to the TypeScript compiler wiki at https://github.com/Microsoft/TypeScript/wiki/Compiler-Options for a full rundown of options available in the compiler API.

Installing TypeScript typings

Besides the project dependencies, such as Bootstrap and Angular 2's own dependencies, TypeScript does require some additional libraries so we can get the best out of it. Specifically, ES6 extends the JavaScript environment with methods and APIs that need to be described to the TypeScript compiler. Otherwise, it will not recognize them as part of the syntax and will throw errors upon compiling. Whenever we need to instruct the TypeScript compiler about a JavaScript API, either a native one or any other API belonging to a third party library, we will want to use a TypeScript type definition file.

A TypeScript type definition file is basically a file with the `d.ts` file extension that contains TypeScript interfaces (more on this in [Chapter 2](#), *Introducing TypeScript*) so we can better perform real-time type checking and prevent compiler errors. Installing type definition files in our projects is not a big deal and just requires having a *typings* tool installed in our environment. In fact, we need to install a type definition file to ensure that the TypeScript compiler is acquainted with the most up-to-date ES6 API. Good news is that we can install a TypeScript definitions manager tool right from the NPM registry, so we can automate the process of searching, installing and deploying type definition files. Therefore, return to the console and proceed with the following commands:

```
$ npm install -g typings
$ typings install es6-shim --ambient --save
```

First, we install the `typings` tool globally and then we leverage the `typings` CLI to install the `es6-shim` types definition file into our project, creating the `typings.json` file that will store the references to the source origin for all type definition files we will install now and later on. A new folder named `typings` is created and it contains the definition files we require. Without them, basic ES6 features like the new functional methods of the `Array` class would not be available.

Before moving forward, we need to tackle one more step regarding the TypeScript typings. When installing type definition files, two façade files are generated by the CLI: `typings/main.d.ts` and `typings/browser.d.ts`. However, only one should be exposed to the TypeScript compiler. Otherwise, it will raise an exception after finding duplicated type definitions. Since we are building frontend applications, we will stick to `browser.d.ts` and exclude `main.d.ts` and its linked definition files by marking it as excluded at `tsconfig.json`:

```json
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "noImplicitAny": false,
    "outDir": "built",
    "rootDir": ".",
    "sourceMap": false
```

```
  },
  "exclude": [
    "node_modules",
    "typings/main.d.ts",
    "typings/main"
  ]
}
```

On the other hand, it is actually recommended to exclude the `typings` folder from your project distribution by including it in your `.gitignore` file, same as we usually do with the `node_modules` folder. You only want to include the `typings.json` manifest when distributing your app and then have all the installation processes handled by `npm`, so it is very convenient to include the type definition files installation as an action handled by the `postinstall` script in the `package.json` file. This way, we can install the `npm` dependencies and the definition files in one shot. The code is as follows:

```
"scripts": {
  "typings": "typings",
  "postinstall": "typings install"
},
```

When taking this approach, the `typings` package should be included in the `package.json` as part of the development dependencies. Thus, reinstall it with the `--save-dev` flag. Again, please refer to the book code repository at GitHub to fetch the latest version of the `package.json` file for this chapter.

Hello, Angular 2!

With the Angular 2 library bundle in place and full support for TypeScript now available, the time has come to put everything to the test. First, create and empty file named `hello-angular.ts` (`.ts` is the natural extension for TypeScript files) at the root of our working folder.

Note

Here, we stumble upon the first of many coding conventions we will cover in this book: file naming. We name our module files using lower kebab case. In Chapter 5, *Building an Application with Angular 2 Components*, we will delve deeper into naming conventions and best practices for coding Angular 2 applications. Until then, we will concur into some anti-patterns for learning purposes, as those more experienced readers will soon notice.

Now, open that file and write the following at the top:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';
```

We have just imported the most basic type and function we will need to scaffold a very basic component in the next section. The importing syntax will be familiar to those who are already familiar with ECMAScript 6. For those who are not familiar with its code paradigm, don't worry. We will discuss more on this in Chapter 2, *Introducing TypeScript*.

TypeScript classes

Let's now define a class:

```
class HelloAngularComponent {
  greeting: string;
  constructor() {
    this.greeting = 'Hello Angular 2!';
  }
}
```

ECMAScript 6 (and TypeScript as well) introduced classes as one of the core parts of its building blocks. Our example features a class field property named `greeting` typed as `string`, which is populated within the constructor with a text string, as you can see in the preceding code. The constructor function is called automatically when an instance of the class is created, and each and every property (and functions as well) should be annotated with the type it represents (or returns in the case of functions).

Do not worry about all this now. [Chapter 2](#), *Introducing TypeScript*, will give you the insights you need to better understand the mechanics of TypeScript. Now, let's focus on the actual layout of our component. You have probably noticed the name structure, which conforms to another common coding convention in Angular 2. We define each and every class in Pascal casing by appending a suffix pointing out its type (will it be a component, directive, pipe, and so on), which is `Component` for this case.

Introducing metadata decorators

The controller class we have just created gives us the machinery we need to instance an object exposing a `greeting` property, but we still need to apply some Angular 2 sugar to turn it into an actual component. We already imported the `Component` metadata class, remember? Let's put it to work and decorate our class like this:

```
@Component({
  selector: 'hello-angular',
  template: '<h1> {{greeting}} </h1>'
})

class HelloAngularComponent {
  greeting: string;
  constructor() {
    this.greeting = 'Hello Angular 2!';
  }
}
```

A decorator is a very interesting experimental feature proposed by ECMAScript 7 that was later embraced and implemented by TypeScript in order to decorate classes with metadata. There are several types of decorators and all of them are easily recognizable by the @ symbol prefix. Although [Chapter 2](#), *Introducing TypeScript,* will give you an idea about decorators, delving deeper into its core logic is out of the scope of this book. However, we will get used to them as we advance through the book.

In this example, we are telling the compiler that the `HelloAngularComponent` class is, in fact, an Angular 2 component. The component is meant to be encapsulated by the `<hello-angular>` custom element and the template property defines the internal HTML structure of our component. As we already mentioned previously, custom elements encapsulating HTML templates are the foundation of web components.

Compiling TypeScript into browser-friendly JavaScript

We are done with our primer on TypeScript, but unfortunately it is quite likely that our browser of choice will not support TypeScript. So, we need to compile our source code into good old ECMAScript 5 JavaScript code.

The good news is that the TypeScript CLI contains tools to compile TypeScript into JavaScript code out of the box. To do this, just open a terminal window and type the following command at the location of the `hello-angular.ts` file:

```
$ tsc --watch
```

A new `hello-angular.js` file will show up within the `built` directory (or the path you have defined in the `outDir` property at `tsconfig.json`), and it will contain an ECMAScript 5 version of the TypeScript code we just built. This file already contains some functional code to implement support for the Metadata decorator we configured.

Tip

Please note the `--watch` flag in our command. This parameter informs the compiler that we want the compilation to be automatically triggered again upon changing any file. Disregard the flag when you just need to compile your stuff once and do not need to watch the code for changes.

Our component is looking better by the minute and now we are in a good state to start using it, but we still need to embed it somewhere in order to see it live. It's time to define the HTML shell or web container where it will live.

The HTML container

Create an HTML file at the root of our workspace and name it `index.html`. Then, populate it with the following code:

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Angular 2!</title>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>

  </head>

  <body>

  </body>

</html>
```

This is the most basic, barebones version of an HTML container for an Angular 2 application we can come up with. This is great because most of the presentation logic and dependency management will be handled by our component itself.

However, two things catch our attention in this template. On one hand, we find a block of script includes. This block contains all the peer dependencies we require to polyfill ES2015 (ES6) functionalities; the Zone and Observables specifications ensure full support for metadata decorators and last, but not least, implement dynamic module loading functionalities to our application.

Note

Do not try to reshuffle the sorting layout of these code blocks unless you want to face unexpected exceptions.

Then, we introduce a script include pointing to a new file named `systemjs.config.js`. This file is yet to be written, so let's create it in the root of our project folder with the following implementation. We will break down this setup in the next paragraphs:

```javascript
(function() {

  var pathMappings = {
    '@angular': 'node_modules/@angular',
    'rxjs': 'node_modules/rxjs',
```

```
    };

    var packages = [
      '@angular/common',
      '@angular/compiler',
      '@angular/core',
      '@angular/http',
      '@angular/platform-browser',
      '@angular/platform-browser-dynamic',
      '@angular/router',
      '@angular/router-deprecated',
      '@angular/testing',
      'rxjs',
      'built',
    ];

    var packagesConfig = {};

    packages.forEach(function(packageName) {
      packagesConfig[packageName] = {
        main: 'index.js',
        defaultExtension: 'js'
      };
    });

    System.config({
      map: pathMappings,
      packages: packagesConfig,
    });

})();
```

As we can see, the `systemjs.config.js` file contains primarily an immediately invoked function expression. This means that this block of code will be executed as soon as it is parsed by the browser. Let's overview each and every piece of this routine:

```
var pathMappings = {
  '@angular': 'node_modules/@angular',
  'rxjs': 'node_modules/rxjs',
};
```

Here, we defined an object literal containing key/value pairs of indexes containing full paths. We will use this object literal to configure the path aliases in SystemJS later on in this same file. Therefore, executing the `import '@angular/core'` statement will instruct SystemJS to actually import the core package from `node_modules/@angular/core`. However, packages are not imported as a whole as is. We need an entry point to such package and, therefore, we need to inform SystemJS what façade file it should seek when importing an entire module. This we will do by defining an array of modules and then creating an object literal, where each module path is the key of a property containing the main entry point and the default file extension configuration object for such paths:

```
var packages = [
  '@angular/common',
```

```
  '@angular/compiler',
  '@angular/core',
  '@angular/http',
  '@angular/platform-browser',
  '@angular/platform-browser-dynamic',
  '@angular/router',
  '@angular/router-deprecated',
  '@angular/testing',
  'rxjs',
  'built',
];

var packagesConfig = {};

packages.forEach(function(packageName) {
  packagesConfig[packageName] = {
    main: 'index.js',
    defaultExtension: 'js'
  };
});
```

The resulting `packagesConfig` variable represents the aforementioned configuration object literal. It also includes main entry file and default extension data for the built folder, which is the folder where the TypeScript compiler will go saving the transpiled files while we develop our application.

With all this configuration in place, our last step is to finally configure SystemJS with these path mappings, entry point, and default file extension expected for each package represented by the previous paths:

```
System.config({
  map: pathMappings,
  packages: packagesConfig,
});
```

This concludes all the configuration required for SystemJS. With this in place, we can both kickstart our application and leverage the ES2015 module import syntax in our code, as we will see in the next sections.

Serving the examples of this book

Before moving on with our example, we need a local web server to execute the examples contained in this book. If you already have a working web server that you can configure to point to your working directory, then skip to the next section. Otherwise, set up a web server in your workspace. As an easy workaround, we recommend you install the extraordinarily powerful and lightweight lite-server node module from NPM:

```
$ npm install -g lite-server
```

Then, you can run a web server with live-reloading functionality by running the following command in a terminal shell after moving into your project folder:

```
$ lite-server
```

After executing the preceding command, a browser window will be fired, pointing to your working directory. Please refer to the NPM module official repository in order to check out all the options available (https://github.com/johnpapa/lite-server).

Tip

It is actually recommended to install the lite-server package paired up with the typescript and concurrently packages, all of them as development dependencies installed with the --save-dev flag. This way, you can run the TypeScript compiler in watch mode and the local server at the same time with a single command that can be wrapped in the start script of package.json. Then, you can start building stuff right away by accessing your working folder and executing npm start. This book's code repository in GitHub implements this approach, so feel free to borrow the package.json example for your convenience.

Putting everything together

Our HTML file is now ready to host our Angular 2 component. To do so, let's edit the template again and drop a custom element with the same tag name we defined in the selector property of our component. Then, import the actual file that contains the component class declaration, leveraging the API of SystemJS for that. Check out these changes in the following example:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello Angular 2!</title>
    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
      // Here we import the component module
      // with no file extension
      System.import ('built/hello-angular');
    </script>
  </head>
  <body>
    <!-- This is our custom element tag -->
    <hello-angular></hello-angular>
  </body>
</html>
```

How cool is that? Now, we can create our own custom elements that render whatever we define in them. Let's bring up the page in our web server and see it in action by going to `http://localhost:3000/` (or whatever host and port your local web server operates in).

Unfortunately, if we reload the browser, nothing will happen and we will only see a blank page with nothing in there. This is because we still need to bootstrap our component to instantiate it on the HTML page.

Let's return to our component file `hello-angular.ts` and add a final line of code:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'hello-angular',
  template: '<h1> {{greeting}} </h1>'
})
class HelloAngularComponent {
  greeting: string;
  constructor() {
    this.greeting = 'Hello Angular 2!';
```

```
  }
}
```

```
bootstrap(HelloAngularComponent); // Component is bootstrapped!
```

The `bootstrap` command instances the controller class we pass as an argument and uses it to lay out a complete application scaffold. Basically, the bootstrap method kickstarts the following actions:

- Analyzes the component configured as its first argument and checks its type.
- Searches the DOM after an element with a tag matching the component selector.
- Creates a child injector that will handle the injection of dependencies in that component and all the child directives (including components, which are directives too) that such a component might host, in a very similar way a tree has ramifications.
- It creates a new Zone. We will not cover Zones in this book, but let's just say that Zones are in charge of managing the change detection mechanism of each instance of a bootstrapped component in an isolated fashion.
- It creates a Shadow DOM context in the custom element identified by the component selector and renders within the HTML defined in the component template.
- The component controller class is instantiated straight away and the change detection machinery is fired. Now that we have the Shadow DOM placeholders in place, data providers are initiated and data is injected where required.

Later in this book, we will cover how we can leverage the `bootstrap` command to display debugging information or how application providers can be globally overridden throughout the whole application so the dependency injector baked in Angular 2 picks the right dependency where required.

Hopefully, you are running the TypeScript compiler in watch mode. Otherwise, please execute the `tsc` command to transpile our code to ES5 and reload the browser. We can delight ourselves with the rendered content of our very first Angular 2 component. Yay!

Enhancing our IDE

Before moving on with our journey through Angular 2, it's time to take a look at IDEs too. Our favorite code editor can become an unparalleled ally when it comes to undertaking an agile workflow entailing TypeScript compilation at runtime, static type checking and introspection, and code completion and visual assistance for debugging and building our app. That being said, let's highlight some major code editors and take a bird's eye view of how each one of them can assist us when developing Angular 2 applications. If you're just happy with triggering the compilation of your TypeScript files from the command line and do not want to have visual code assistance, feel free to skip to the next section. Otherwise, jump straight to the following section that covers the IDE of your choice.

Sublime Text 3

This is probably one of the most widespread code editors nowadays, although it has lost some momentum lately with users favoring other rising competitors such as GitHub's very own Atom. If this is your editor of choice, we will assume that it's already installed on your system and you also have Node (which is obvious, otherwise, you could have not installed TypeScript in first place through NPM). In order to provide support for TypeScript code editing, you need to install Microsoft's TypeScript plugin, available at https://github.com/Microsoft/TypeScript-Sublime-Plugin. Please refer to this page to learn how to install the plugin and all the shortcuts and key mappings.

Once successfully installed, it only takes *Ctrl* + space bar to display code hints based on type introspection (see the following screenshot). On top of that, we can trigger the build process and compile the file to the JavaScript we are working on by hitting the *F7* function key. Real time code error reporting is another fancy functionality you can enable from the command menu.
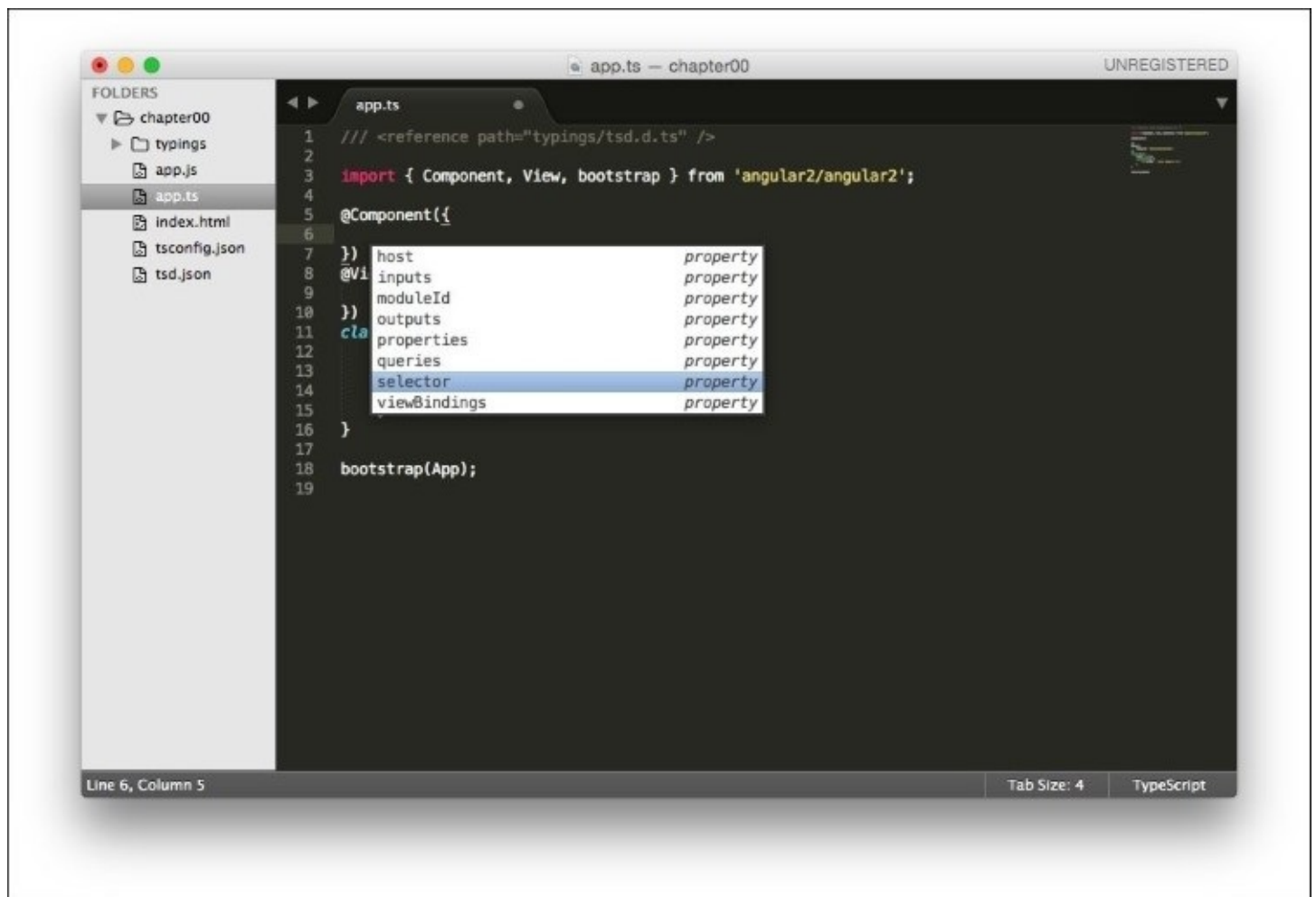
Atom

Developed by GitHub, the highly customizable environment and ease of installation of new packages has turned Atom into the IDE of choice for a lot of people. It is worth mentioning that the code examples provided in this book were actually coded using Atom only.

In order to optimize your experience with TypeScript when coding Angular 2 apps, you need to install the Atom TypeScript package. You can install it by means of the APM CLI or just use the built-in package installer. The functionalities included are pretty much the same as we have in Sublime after installing the Microsoft package: automatic code hints, static type checking, code introspection, or automatic build upon save to name a few. On top of that, this package also includes a convenient built-in tsconfig.json generator.
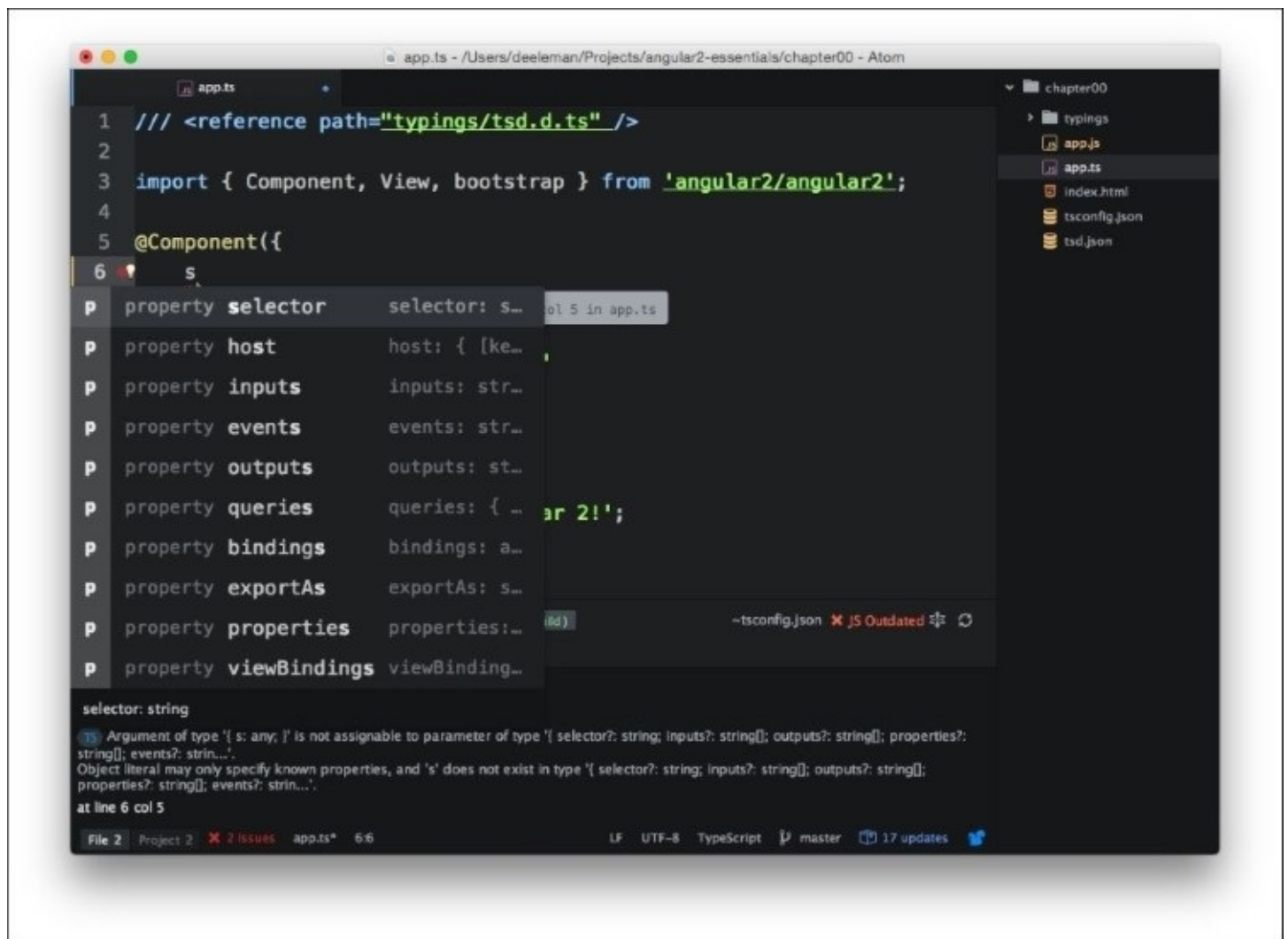
Visual Studio Code

Visual Studio Code, a relatively new code editor backed by Microsoft, is gaining momentum as a serious contender in the Angular 2 medium, mostly because of its great support for TypeScript out of the box. TypeScript has been, to a greater 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. This means that all the nice features we might want are already baked in, including syntax and error highlighting and automatic builds.
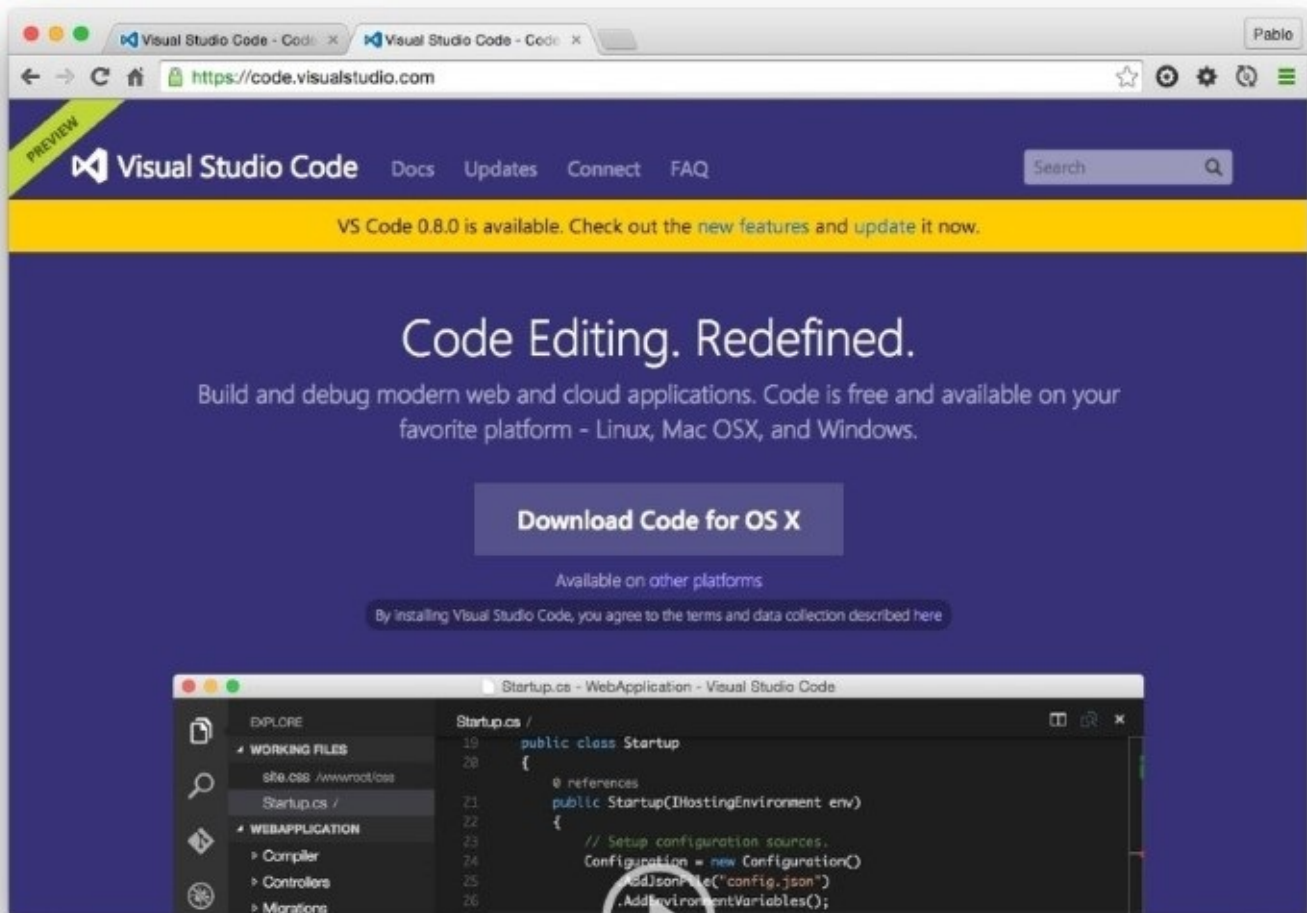
WebStorm

This excellent code editor supplied by IntelliJ is also a great pick for coding Angular 2 apps based on TypeScript. The IDE comes with built-in support for TypeScript out of the box so that we can start developing Angular 2 components from day one. WebStorm also implements a built-in transpiler with support for file watching, so we can compile our TypeScript code into pure vanilla JavaScript without relying on any third-party plugins.

Leveraging Gulp with other IDEs

May be your IDE is not listed here and you do not want to switch from your favorite code editor now, not having the chance, for whatever reason, to automate TypeScript compilation for your project. Or perhaps you do not feel very comfortable messing around with the TypeScript commands on the console.

If this is not the case, feel free to skip to the next section. However, if you relate to this case scenario, don't worry. Modern JavaScript task runners have your back. Let's pick Gulp (http://gulpjs.com) and see how we can create a super simple script to automate TypeScript compilation in our project.

First, let's proceed with the dependencies installation. Basically, we will install Gulp and then `gulp-typescript`, a typescript compiler for gulp with incremental compilation support. On your console window, type the following commands:

```
$ npm install -g gulp
$ npm install gulp gulp-typescript --save-dev
```

Let's create a JavaScript file named `gulpfile.js` at the root of your project with the following content:

```
var gulp = require('gulp');
var ts = require('gulp-typescript');
var tsProject = ts.createProject('tsconfig.json');

gulp.task('build', function() {
  var tsResult = tsProject.src().pipe(ts(tsProject));
  return tsResult.js.pipe(gulp.dest('./built'));
});
```

You will need a `tsconfig.json` file at the root of your project, so our Gulp task can fetch our compilation preferences from it. From this moment onwards, we can launch the build processing over the files listed at the files' array property in our `tsconfig.json` file by executing the following command:

```
$ gulp build
```

Unfortunately, the `gulp-typescript` plugin does not support file watching, so if we want to trigger the build processing automatically every time a TypeScript file change, we need to rely on Gulp's native `watch` method. To do so, just add the following chunk of code at the end of our `gulpfile.js` file:

```
gulp.task('watch', ['build'], function() {
  gulp.watch('./**/*.ts', ['build']);
});
```

Now, you can launch the build process and watch the file changes by executing the following command:

```
$ gulp watch
```

Diving deeper into Angular 2 components

We have come a long way now, from tapping on TypeScript for the first time to learning how to code the basic scripting schema of an Angular 2 component. However, before jumping onto more abstract topics, let's flesh out our current component with more functionalities and take an overview of the most common traits of Angular apps and components.

Improving productivity

Sometimes, we need some helpers to boost our focus, especially when we deal with too abstract stuff that requires additional attention. A widely accepted approach is the *Pomodoro technique,* in which we put together a task list and then split the deliverables into to-do items that won't require us more than 25 minutes to accomplish. When we pick any of those to-do items, we focus under distraction-free mode on its delivery for 25 minutes with the help of a countdown timer. You can grab more information about this technique at http://pomodorotechnique.com.

In this book, we are going to build a major component that represents this functionality and fill the component with additional functionalities and UI items wrapped inside their own components. To do so, we will use the Pomodoro technique, so let's start by creating a Pomodoro timer.

Component methods and data updates

Create a new `pomodoro-timer.ts` file in the same folder and populate it with the following basic implementation of a very simple component. Don't worry about the added complexity, we will review each and every change made after the code block:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'pomodoro-timer',
  template: '<h1> {{ minutes }}:{{ seconds }} </h1>'
})
class PomodoroTimerComponent {
  minutes: number;
  seconds: number;

  constructor() {
    this.minutes = 24;
    this.seconds = 59;
  }
}

bootstrap(PomodoroTimerComponent);
```

Our new component is, in fairness, not that much different from the one we previously had. We updated the names to make them more self-descriptive and then defined two property fields, statically typed as numbers in our `PomodoroTimerComponent` class. These are rendered in the contained view, wrapped inside an `<h1>` element. Now, open the `index.html` file and replace the `<hello-angular></hello-angular>` custom element with our new `<pomodoro-timer></pomodoro-timer>` tag. You can duplicate `index.html` and save it under a different name if you do not want to loose the HTML side of our fancy "Hello World" component.

Note

A note about naming custom elements

Selectors in Angular 2 are case sensitive. As we will see later in this book, components are a sub set of directives that can support a wide range of selectors. When creating components, we are supposed to set a custom tag name in the selector property by enforcing a dash-casing naming convention. When rendering that tag in our view, we should always close the tag as a non-void element. So `<custom-element></custom-element>` is correct, while `<custom-element />` will trigger an exception. Last but not least, certain "common" camel case names might conflict with the Angular 2 implementation, so avoid names like `AppView` or `AppElement`.

You will want to update the reference in your `System.import(...)` block at `index.html` to point to our new component as well:

```
System.import('built/pomodoro-timer').then(null, console.error.bind(console));
```

Now, it is a good time to mention that the `import` method of SystemJS is asynchronous and returns a promise once the module has been successfully loaded. We can leverage this promise to throw any eventual error message to the console, which will become quite handy whenever we have to debug our code. You will see this practice later in this book.

If you bring up a browser window and load this file, you will see a representation of the numbers defined in the component class. But we want to do more than just display a handful of numbers, right? We actually want them to represent a time countdown, and we can achieve that by introducing these changes. Let's first introduce a function we can iterate on in order to update the countdown. Add this function after the constructor function:

```
tick(): void {
  if (--this.seconds < 0) {
    this.seconds = 59;
    if (--this.minutes < 0) {
      this.minutes = 24;
      this.seconds = 59;
    }
  }
}
```

As you can see here, functions in TypeScript need to be annotated with the type of the value they return, or just `void` if none. Our function assesses the current value of both `minutes` and `seconds`, and then either decreases their value or just resets it to the initial value. Then this function is called every second by triggering a time interval from the class constructor:

```
constructor() {
  this.minutes = 24;
  this.seconds = 59;
  setInterval(() => this.tick(), 1000);
}
```

Here, we spot for the first time in our code an arrow function (also known as a lambda function, fat arrow, and so on), a new syntax for functions brought by ECMAScript 6 that we will cover in more detail in [Chapter 2](#), *Introducing TypeScript*. The `tick` function is also marked as `private`, so it cannot be inspected or executed outside a `PomodoroTimerComponent` object instance.

So far so good! We have a working Pomodoro timer that countdowns from 25 minutes to 0, and then starts all over again. The problem is that we are replicating code here and there. So, let's refactor everything a little bit to prevent code duplication.

```
constructor() {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);
}

resetPomodoro(): void {
```

```
    this.minutes = 24;
    this.seconds = 59;
}

private tick(): void {
  if (--this.seconds < 0) {
    this.seconds = 59;
    if (--this.minutes < 0) {
      this.resetPomodoro();
    }
  }
}
```

We have wrapped the initialization (and reset) of minutes and seconds inside our function `resetPomodoro`, which is called upon instantiating the component or reaching the end of the countdown. Wait a moment though! According to the Pomodoro technique, pomodoro practitioners are allowed to rest in between pomodoros or even pause them should an unexpected circumstance gets on the way. We need to provide some sort of interactivity so the user can start, pause, and resume the current pomodoro timer.

Adding interactivity to the component

Angular 2 provides a top-notch support for events through a declarative interface that reminds the one in other frameworks such as React. Let's first modify our template definition:

```
@Component({
  selector: 'pomodoro-timer',
  template: `
    <h1> {{ minutes }}:{{ seconds }} </h1>
    <p>
      <button (click)="togglePause()">
        {{ buttonLabel }}
      </button>
    </p>
  `
})
```

We used a multiline text string! ECMAScript 6 introduced the concept of *template strings*, which are string literals with support for embedded expressions, interpolated text bindings, and multiline content. We will look into them in more detail in [Chapter 2](), *Introducing TypeScript*.

In the meantime, just focus on the fact that we introduced a new chunk of HTML that contains a button with an event handler that listens to click events and executes the `togglePause` method upon clicking. This `(click)` attribute is something you might not have seen before, even though it is fully compliant with the W3C standards. Again, we will cover this in more detail in [Chapter 3](), *Implementing Properties and Events in Our Components*. Let's focus on the `togglePause` method and the new `buttonLabel` binding. First, let's modify our class properties so that they look like this:

```
class PomodoroTimerComponent {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabel: string;

  // … Rest of code will remain as it is below this point
}
```

We introduced two new fields. First is `buttonLabel` that contains the text that will be later on displayed on our newly-created button. `isPaused` is a newly-created variable that will assume a true/false value, depending on the state of our timer. So, we might need a place to toggle the value of such a field. Let's create the `togglePause` method we mentioned earlier:

```
togglePause(): void {
  this.isPaused = !this.isPaused;
  // if countdown has started
  if (this.minutes < 24 || this.seconds < 59) {
    this.buttonLabel = this.isPaused ? 'Resume' : 'Pause';
  }
}
```

In a nutshell, the `togglePause` method just switches the value of `isPaused` to its opposite and then, depending on such new value and whether the timer has started (which would entail that any of the time variables has a value lower than the initialization value) or not, we assign a different label to our button.

Now, we need to initialize these values, and it seems there is no better place for it. So, the `resetPomodoro` function is the place where variables affecting the state of our class are initialized:

```
resetPomodoro(): void {
  this.minutes = 24;
  this.seconds = 59;
  this.buttonLabel = 'Start';
  this.togglePause();
}
```

By executing `togglePause` every time, we reset the Pomodoro to make sure that whenever the Pomodoro reaches a state where it requires to be reset, the countdown behavior will switch to the opposite state it had previously. There is only one tweak left in the controller method that handles the countdown:

```
private tick(): void {
  if (!this.isPaused) {
    this.buttonLabel = 'Pause';

    if (--this.seconds < 0) {
      this.seconds = 59;
      if (--this.minutes < 0) {
        this.resetPomodoro();
      }
    }
  }
}
```

Obviously, we do not want the countdown to continue when the timer is supposed to be paused, so we wrap the whole script in a conditional. In addition to this, we will want to display a different text on our button whenever the countdown is not paused and once again when the countdown reaches its end, stopping and then resetting the pomodoro to its initial values will be the expected behavior. This reinforces the need of invoking the `togglePause` function within `resetPomodoro`.

Improving the data output in the view and polishing the UI

So far, we reloaded the browser and played around with the newly created toggle feature. However, there is apparently something that still requires some polishing: when the seconds counter is less than 10, it displays a single-digit number instead of the usual two-digit numbers we are used to see in digital clocks and watches. Luckily, Angular 2 implements a set of declarative helpers that format the data output in our templates. We call them *Pipes*, and we will cover them in detail later in [Chapter 3](#), *Implementing Properties and Events in Our Components*. For the time being, let's just introduce the number pipe in our component template and configure it to format the seconds output to display two digits all the times. Update our template so that it looks like this:

```
@Component({
  selector: 'pomodoro-timer',
    template: `
      <h1> {{ minutes }}:{{ seconds | number: '2.0' }} </h1>
      <p>
        <button (click)="togglePause()">
          {{ buttonLabel }}
        </button>
      </p>
    `

})
```

Basically, we appended the pipe name to the interpolated binding in our template separated by a pipe (|) symbol, hence the name. Reload the template and you will see how the seconds figure always displays two digits, regardless of the value it assumes.

We have created a fully functional Pomodoro Timer widget that we can reuse or embed in more complex applications. [Chapter 5](#), *Building an Application with Angular 2 Components*, will guide us through the process of embedding and nesting our components in the context of larger component trees.

In the meantime, let's add some UI beautification to make our component more appealing. We already introduced a class attribute in our button tag as an anticipation of the implementation of the Bootstrap CSS framework in our project. Let's import the actual stylesheet we downloaded through NPM when installing the project dependencies. Open pomodoro-timer.html and add this snippet at the end of the <HEAD> element:

```
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.min.css">
```

Now, let's beautify our UI by inserting a nice page header right before our component:

```
<body>
  <nav class="navbar navbar-default navbar-static-top">
    <div class="container">
      <div class="navbar-header">
        <strong class="navbar-brand">My Pomodoro Timer</strong>
```

```
      </div>
    </div>
  </nav>
  <pomodoro-timer></pomodoro-timer>
</body>
```

Tweaking the component button with a Bootstrap button class will give it more personality and wrapping the whole template in a centering container and appending a nice icon at the top will definitely compound up the UI. So let's update the template in our template to look like this:

```
<div class="text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro">
  <h1> {{ minutes }}:{{ seconds  | number: '2.0' }} </h1>
  <p>
    <button (click)="togglePause()"
      class="btn btn-danger">
      {{ buttonLabel }}
    </button>
  </p>
</div>
```

For the icon, we picked a bitmap icon depicting a pomodoro. You can use any bitmap image of your choice or you can just skip the icon for now, even though we will need an actual pomodoro icon in the forthcoming chapters. This is how our Pomodoro timer app looks after implementing all these visual tweaks:

Tip

Downloading the example code

You can download the example code files for this book from GitHub at
https://github.com/deeleman/learning-angular2.

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

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the **SUPPORT** tab at the top.
- Click on **Code Downloads & Errata**.
- Enter the name of the book in the **Search** box.

- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

Summary

We looked at web components according to modern web standards and how Angular 2 components provide an easy and straightforward API to build our own components. We covered TypeScript and some basic traits of its syntax as a preparation for [Chapter 2](), *Introducing TypeScript*. We saw how to set up our working space and where to go to find the dependencies we need to bring TypeScript into the game and use the Angular 2 library in our projects, going through the role of each dependency in our application.

Our first component gave us the opportunity to discuss the form of a controller class containing property fields, constructor, and utility functions, and why metadata annotations are so important in the context of Angular 2 applications to define how our component will integrate itself in the HTML environment where it will live. Now, we also know how to deploy web server tools and enhance our code editors to make our lives easier when developing Angular 2 apps, leveraging type introspection and checking on the go. Our first web component features its own template and such templates host property bindings declaratively in the form of variable interpolations, conveniently formatted by pipes. Binding event listeners is now easier than ever and its syntax is standards-compliant.

The next chapter will cover, in detail, all the TypeScript features we need to know to get up to speed with Angular 2 in no time.

Chapter 2. Introducing TypeScript

In the previous chapter, we built our very first component and we used TypeScript to shape the code scripts, which gave form to it. All the examples included in this book use its syntax. As we will see later in this book, writing our scripts in TypeScript and leveraging its static typing will give us a remarkable advantage over the other scripting languages.

This chapter is not a thorough overview of the TypeScript language. We will just focus on the core elements of the language and study them in detail on our journey through Angular 2. 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 will:

- Look at the background and rationale behind TypeScript
- Discover online resources to practice while we learn
- Recap on the concept of typed values and how to represent them
- Build our own types, based on classes and interfaces
- Learn to better organize our application architecture with modules

Understanding the case for TypeScript

The natural evolution of the early JavaScript-driven small web applications into thick monolithic clients unveiled the shortcomings of the ECMAScript 5 JavaScript specification. In a nutshell, large-scale JavaScript applications suffered from serious maintainability and scalability problems as soon as they grew up in size and complexity.

This issue became more relevant as new libraries and modules required seamless integration onto our applications. The lack of good mechanisms for interoperability led to really cumbersome solutions that never seemed to fit the bill.

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

Microsoft took the challenge and spent nearly 2 years building a superset of the language, combining the conventions of ES6 and borrowing some proposals from ES7. The idea was to launch something that helped out with building enterprise applications with a lower error footprint by means of static type checking, better tooling, and code analysis.

After 2 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 it reached Version 1.0 two years later. TypeScript was not only running ahead of ECMAScript 6, but it also implemented the same features and provided a solid environment for building large-scale applications by introducing, among other features, optional static typing through type annotations, thereby ensuring type checking at compile time. This contributes to catching errors in earlier stages of the development process. The support for declaration files also gives developers the opportunity to describe the interface of their modules, so other developers can better integrate them into their code workflow and tooling.

The benefits of TypeScript

The following infographic provides a bird's eye view of the different features that distinguishes ECMAScript 6 from ECMAScript 5, and then differentiates TypeScript from the two.

ECMAScript 5

ECMAScript 6

classes, inheritance
arrow functions
template strings
let, const, destructuring

default, rest, spread arguments
modules, loaders
promises
etc...

TypeScript

type annotations
type checking
introspection
interfaces

enums
decorators
generics
etc...
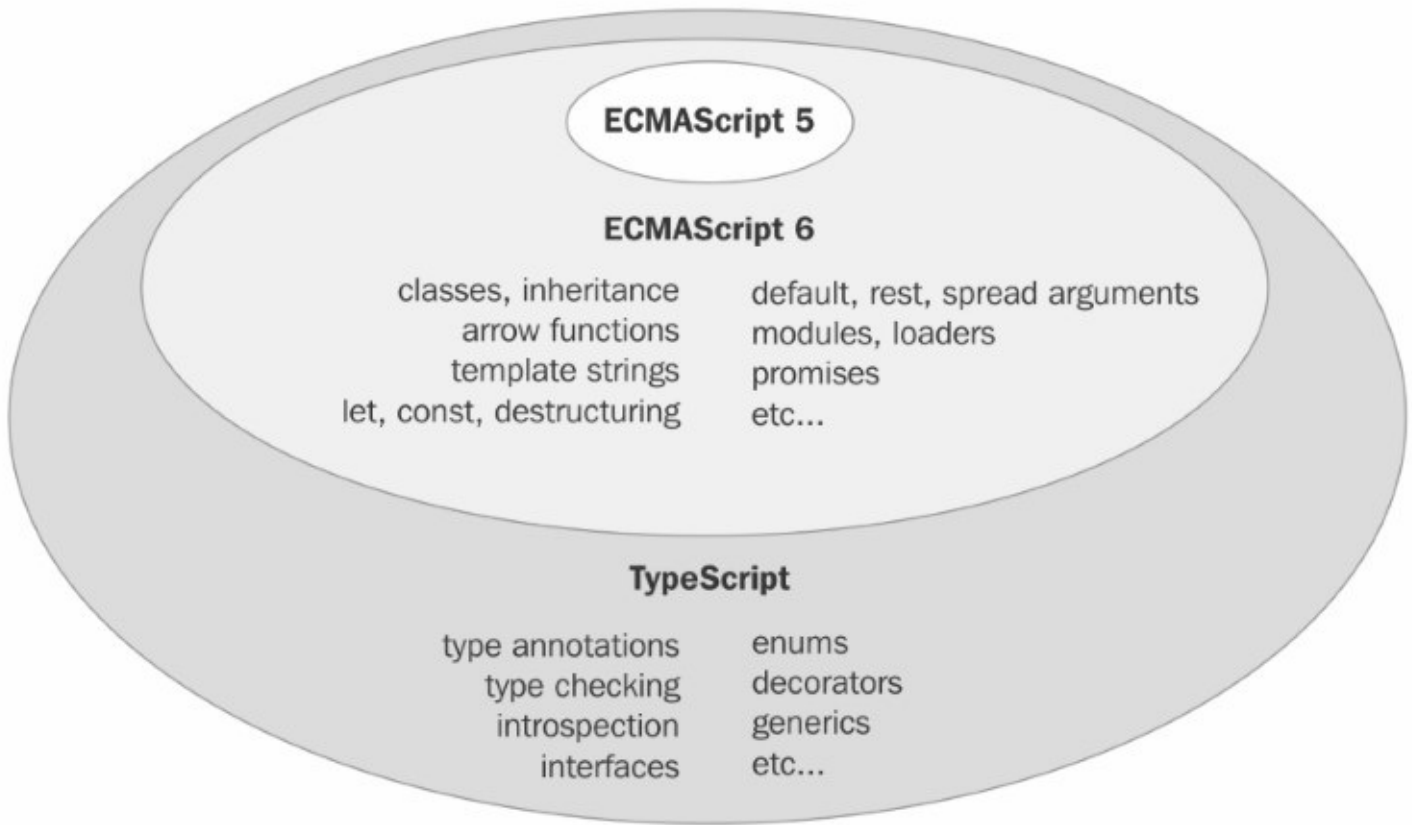
As a superset of ECMAScript 6, one of the main advantages of embracing TypeScript in your next project is the low entry barrier. If you know ECMAScript 6, you are pretty much all set, since all the additional features in TypeScript are optional. You can pick and introduce in your practice the features that help you to achieve your goal. All in all, there is a long list of strong arguments for advocating for TypeScript in your next project and all of them obviously apply to Angular 2 as well. Here is a short rundown of arguments, just to name a few:

- Annotating our code with types ensures a consistent integration of our different code units and improves code readability and comprehension.
- The TypeScript's built-in type-checker will analyze your code at runtime and help you prevent errors even before executing your code.
- The use of types ensures consistency across your applications. In combination with the previous two, the overall code errors footprint gets minimized in the long run.
- TypeScript extends classes with longtime demanded features such as class fields, private members, enums, and so on.
- The use of decorators opens the door to extend our classes and implementations in

unparalleled ways.

- Creating interfaces and type definition files (which we will not cover in this book though) ensures a smooth and seamless integration of our libraries in other systems and codebases.
- TypeScript support across the different IDEs on store is terrific, and we can benefit from code highlighting, real-time type checking, and automatic compilation at no cost.
- The TypeScript syntax will definitely please developers coming from other backgrounds such as Java, C#, C++, and so on.

Introducing TypeScript resources in the wild

In the previous chapter, we saw how to install TypeScript in our system and use the compiler for transpiling our TypeScript files into ES5 script files. Now, we are going to take a look at where can we get further support to learn and test-drive our new knowledge of TypeScript.

The TypeScript official site

Obviously, our first stop is the official site for the language: http://www.typescriptlang.org. There, we can find a more extensive introduction to the language and links to IDEs and corporate supporters of this project. Nevertheless, the most important sections that we will definitely revisit more often are the learn section and the play sandbox.

The **learn** section gives us access to a quick tutorial to get up to speed with the language in no time. It might be interesting as a recap on what we discussed in the previous chapter, but we would suggest you to skip it in favor of the sample pages and the language spec, the latter being a direct link to the full extensive documentation of the language at GitHub. This is a

priceless resource for both new and experienced users.

The **play** section offers a convenient sandbox, including some readymade code examples, covering some of the most common traits of the language. We encourage you to leverage this tool to test out the code examples we will see throughout this chapter.

The TypeScript Wiki

We made a reference to the TypeScript Wiki in the previous chapter when speaking about the most basic parameters we need to know when executing commands with the TypeScript compiler API.

The code for TypeScript is fully open sourced at GitHub, and the Microsoft team has made a good effort at documenting the different facets of the code in the Wiki available on the repository site. We encourage you to go take a look at it any time you have a question or want to delve deeper into any of the language features or form aspects of its syntax.

The Wiki is located at https://github.com/Microsoft/TypeScript/wiki.

Types in TypeScript

Working with TypeScript or any other coding language means basically working with data, and such data can represent different sorts of content. This is what we know as types, a noun 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. This is nothing new to JavaScript, since we have always been working implicitly with types, but in a flexible manner. This means that any given variable could assume (or return in the case of functions) any type of value. Sometimes, this led 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. While this flexibility can still be enforced by means of any type that we will see later on in this chapter, statically typing our variables gives us and our IDEs a good picture of what kind of data we are supposed to find on each instance of code. This becomes an invaluable way to help debug our applications at compile time before it's too late.

String

Probably one of the most widely used primitive types in our code will be the `string` type, where we populate a variable with a piece of text:

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

Check out the type assignation next to the variable name, which is separated by a colon symbol. This is how we annotate types in TypeScript, as we already saw in the previous chapter.

Back to the `string` type, we can use either single or double quotes, and it is same as ECMAScript6. We can define multiline text strings with support for text interpolation with placeholder variables by using the same type:

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

Declaring our variables the ECMAScript 6 way

TypeScript, as a superset of ECMAScript 6, supports expressive declaration nouns such as `let`, which informs us that the variable is scoped to the nearest enclosing block (either a function for loop or any enclosing statement). On the other hand, `const` is an indicator that the values declared this way are meant to feature always the same type or value once populated. For the rest of this chapter, we will enforce the traditional `var` notation for declaring variables, but remember that

Number

Number is probably the other most widespread primitive data type along with `string` and `boolean`. The same as in JavaScript, number defines a floating point number. The `number` type also defines hexadecimal, decimal, binary, and octal literals:

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

Boolean

The `boolean` type defines data that can be True or False, representing the fulfillment of a condition:

```
var isZeroGreaterThanOne: boolean = false;
```

Array

Assigning wrong member types to arrays and handling exceptions that arise by that can be now easily avoided with the `Array` type, where we describe an array containing certain types only. The syntax just requires the postfix `[]` in the type annotation, as follows:

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

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

Dynamic typing with the any type

Sometimes, it is hard to infer the data type out of the information we have at some point, especially when we are porting legacy code to TypeScript or integrating loosely typed third-party libraries and modules. Don't worry, 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 on. This great power comes with a great responsibility though. If we bypass the convenience of static type checking, we are opening the door to type errors when piping data through our modules, and it will be up to us to ensure type safety throughout our application:

```
var distance: any;

// Assigning different value types is perfectly fine
distance = '1000km';
distance = 1000;

// Allows us to seamlessly combine different types
var distances: any[] = ['1000km', 1000];
```

The null and undefined JavaScript literals require special mention. In a nutshell, they are typed under the any type. This makes it possible later on to assign these literals to any other variable, regardless its original type.

Enum

Enum is basically a set of unique numeric values that we can represent by assigning friendly names to each one of them. The use of enums goes beyond assigning an alias to a number. We can use them as a way to list, in a convenient and recognizable way, the different variations that a specific type can assume.

Enums are declared using the enum keyword, without var or any other variable declaration noun, and they begin numbering members starting at 0 unless explicit numeric values are assigned to them:

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

Inspecting the value of myCar will return 1 (which is the index held by Cadillac in the enum). As we mentioned already, we can assign custom numeric values in the enum:

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

Inspecting myTruck will yield 2, since the first enumerated value was set as 1 already. We can extend value assignation to all the enum members as long as such values are integers:

```
enum StackingIndex {
  None = 0,
```

```
  Dropdown = 1000,
  Overlay = 2000,
  Modal = 3000
};
var mySelectBoxStacking: StackingIndex = LayerStackingIndex.Dropdown;
```

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

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

Void

The `void` type definitely represents the absence of any type and its use is constrained to annotating functions that do not return an actual value. Therefore, there is no return type either. We already had the chance to see this with an actual example in the previous chapter:

```
resetPomodoro(): void {
  this.minutes = 24;
  this.seconds = 59;
}
```

Type inference

Typing our data is optional since TypeScript is smart enough to infer the data type of our variables and function return values out of context with a certain level of accuracy. When no type inference is possible, TypeScript will assign the dynamic any type to the loosely typed data at the cost of reducing type checking to a bare minimum.

However, it is always a good practice to ensure that the information we handle is conveniently described by explicitly annotating its type, so we can harness the benefit of having the compiler verifying correctness throughout our code.

Functions, lambdas, and execution flow

The same as in JavaScript, functions are the processing machines where we analyze input, digest information, and apply the necessary transformations to the data provided to either transform the state of our application or 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 for the fact that functions, just as everything else in TypeScript, can be annotated with static types and thus, they better inform the compiler of the information they expect in their signature and the data type they aim to return, if any.

Annotating types in our functions

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

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

We can clearly see two main differences from the usual function syntax in regular JavaScript. First, we annotate with type information the parameters declared in the function signature. This makes sense since the compiler will want to check whether the data provided when executing the function holds the correct type. In addition to this, we also annotate the type of the returning value by adding the postfix `string` to the function declaration. In these cases, where the given function does not return any value, the type annotation void will give the compiler the information it requires to provide a proper type checking.

As we mentioned in the previous section, the TypeScript compiler is smart enough to infer types when no annotation is provided. In this case, the compiler will look into the arguments provided and the return statements to infer a returning type from it.

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

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

However, there is a downside of this syntax. Although typing function expressions this way is allowed, thanks to type inference, the compiler is missing the type definition in the declared variable. We might assume that the inferred type of a variable that points to a function typed as `string` is obviously a string. Well, it's not. A variable that points to an anonymous function ought to be annotated with a `function` type. Basically, the `function` type informs about both the types expected in the function payload and the type returned by the function execution, if any. This whole block, in the form of `(arguments: type) => returned type`, becomes the type annotation our compiler expects:

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

Why such a cumbersome syntax, you might ask? Sometimes, we will declare variables that might depend on factories or function bindings. Then, it is always a good practice to provide as much information to the compiler as we can. This simple example might help you to understand better:

```
// Two functions with the same typing but different logic
function sayHello(input: string): string {
    return 'Hello ' + input;
```

```
}

function sayHi(input: string): string {
  return 'Hi ' + input;
}

// Here we declare the variable with its function type
var greetMe: (name: string) => string;

// Last, we assign a function to the variable
greetMe = sayHello;
```

This way, we also ensure that later function assignations conform to the type annotations set when declaring variables.

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. In other words, we cannot declare parameters and then do not included in the payload when executing the function afterwards. The same applies to even to those arguments in the function payload, which were not originally declared and annotated when defining the function. We obviously need a way to cope with this case scenario, so TypeScript offers this functionality by adding the ? symbol as a postfix to the parameter name we want to make optional:

```
function greetMe(name: string, greeting?: string): string {
    if (!greeting) {
        greeting = 'Hello';
    }
    return greeting + ', ' + name;
}
```

Note

When a parameter is marked as optional and not provided when executing the function, TypeScript will assign the `null` value to it. On the other hand, the rule of thumb is to put required parameters first and then optional parameters last.

Default parameters

TypeScript gives us another feature to cope with the scenario depicted earlier in the form of default parameters, where we can set a default value the parameter will assume when not explicitly populated upon executing the function. The syntax is pretty straightforward as we can see when we refactor the previous example here:

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

Just as with optional parameters, default parameters must be put right after the non-default parameters in the function signature.

Rest parameters

One of the big advantages of the flexibility of JavaScript when defining functions is the functionality to accept an unlimited non-declared array of parameters in the form of the `arguments` object. In a statically typed context such as TypeScript, this might be not possible, but it actually is by means of the Rest parameter's object. Here, we can define, at the end of the arguments list, an additional parameter prefixed by ellipsis and typed as an array:

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

alert(greetPeople('Hello', 'John', 'Ann', 'Fred'));
```

Note

It's important to note that the Rest parameters must be put at the end of the arguments list and can be left off whenever not required upon executing the function.

Overloading the function signature

Method and function overloading is a common pattern 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 this functionality out of the box. So, the only workaround possibly requires writing function declarations for each of the overloads and then writing a general-purpose function that will wrap the actual implementation and whose list of typed arguments and returning types are compatible with all the others:

```
function hello(name: string): string;
function hello(names: string[]): string;
function hello(names: any, greeting?: string): string {
  var 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 are exposing three different function signatures and each of them features different type annotations. We could even define different returning types if there was a case for that. For doing so, we should have just annotated the wrapping function with an any return type.

Better function syntax and scope handling with lambdas

ECMAScript 6 introduced the concept of fat arrow functions (also called lambda functions in other languages such as Python, C#, Java, or C++) as a way to both simplify the general function syntax and also to provide a bulletproof way to handle the scope of the functions that are traditionally handled by the infamous scope issues of tackling with the `this` keyword.

The first impression is its minimalistic syntax, where, most of the time, we will see arrow functions as single-line, anonymous expressions:

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

The 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 just need to wrap them all between braces:

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

This makes this syntax extremely convenient when developing functional operations such as `map`, `reduce`, and others:

```
var reducedArray = [23, 5, 62, 16].reduce((a, b) => a + b, 0);
```

Arrow functions can also contain statements. In that case, we will want to wrap the whole implementation in curly braces:

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

Still, what does this have to do with scope handling? Basically, the value of this can point to a different context, depending on where we execute the function. This is a big deal for a language that prides itself on an excellent flexibility for functional programming, where patterns such as callbacks are paramount. When referring to `this` inside a callback, we lose track of the upper context and that usually forces us to use conventions such as assigning the value of `this` to a variable named `self` or `that`, which will be used later on within the callback. Statements containing interval or timeout functions make a perfect example of this:

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

var greeting = new delayedGreeting('Peter')
greeting.greet(); // alerts 'Hello undefined'
```

When executing the preceding script, we won't get the expected `Hello Peter` alert, but an incomplete string highlighting a pesky greeting to *Mr. Undefined!* Basically, this construction screws the lexical scoping of `this` when evaluating the function inside the timeout call. Porting this script to arrow functions will do the trick though:

```
function delayedGreeting(name): void {
    this.name = name;
    this.greet = function() {
      setTimeout(() => alert('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 lexical scoping of `this` will keep pointing to the proper context outside the `setTimeout` call, allowing a more elegant and clean syntax.

Classes, interfaces, and class inheritance

Now that we have overviewed the most relevant bits and pieces of TypeScript, it's time to see how everything falls into place to build TypeScript classes. These classes are the building blocks of TypeScript and Angular 2 applications.

Although the noun `class` was a reserved word in JavaScript, the language itself never had an actual implementation for traditional POO-oriented classes as other languages such as Java or C# did. JavaScript developers used to mimic this kind of functionality, leveraging the function object as a constructor type, which would be later on instanced with the new operator. Other common practices such as extending our function objects were implemented by applying prototypal inheritance or by using composition.

Now we have an actual class functionality, which is flexible and powerful enough to implement the functionality our applications require. We already had the chance to tap into classes in the previous chapter. Let's look at them in more detail now.

Anatomy of a class – constructors, properties, methods, getters, and setters

The following piece of code illustrates how a class could be. Please note that the `class` property members come first and then we include a constructor and several methods and property accessors. None of them features the reserved word `function` and all the members and methods are properly annotated with a type except `constructor`:

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

  constructor(public 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;
  }
}
```

This `class` layout will probably remind us of the component class we built back in Chapter 1, *Creating Our Very First Component in Angular 2*. Basically, the `class` statement wraps several elements that we can break down into:

- **Members**: Any instance of the `Car` class will feature two properties: `color` typed as string, and `distanceRun` typed as a number and only accessible from within the class itself. If we instance 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 function is executed right away when an instance of the class is created. Usually, we want to initialize the class members here, with the data provided in the constructor signature. We can also leverage the constructor signature itself to declare class members, as we did with the `isHybrid` property. To do so, we just need to prefix the constructor parameter with an access modifier such as `private` or `public`. Same as we saw when analyzing functions in the previous sections, we can define *rest, optional*, or *default* parameters as depicted in the previous example with the `color` argument, which fallbacks to "red" when not explicitly defined.
- **Methods**: A method is a special kind of member which represents a function and

therefore, can return, or not, a typed value. Basically, it is a function that becomes part of the object API. Methods can be private as well. In that case, they are basically used as helper functions 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. This means that we can consume static members directly, without having to instantiate an object first. In fact, static members are not accessible from the object instances and thus, they cannot access other class members using `this`. These members are usually included in the class definition as helper or factory methods in order to provide a generic functionality not related to any specific object instance.
- **Property accessors**: In ES5, we could define custom setters/getters in a very verbose way with `Object.defineProperty`. Now, things have become quite simpler. In order to create property accessors (usually pointing to internal private fields as in the example provided), we just need to prefix a typed method named as the property we want to expose with `set` (in order to make it writable) and `get` (in order to make it readable).

As a personal exercise, why don't you copy the preceding piece of code at the playground page (http://www.typescriptlang.org/Playground) and execute it? We can even see an instance object of the `Car` class in action by appending this snippet right after the class definition and running the code and inspecting the output in the browser's developer tools console:

```
var myCar = new Car(false);

console.log(myCar.color);       // 'red'
// Public accessor returns distanceRun:
console.log(myCar.distance);  // 0

myCar.drive(15);
console.log(myCar.distance);  // 15 (0 + 15)
myCar.drive(21);
console.log(myCar.distance);  // 36 (15 + 21)

// What's my carbon footprint according to my car type?
myCar.getGasConsumption();     // 'Too high!'

Car.honk();    // 'HOOONK!' no object instance required
```

We can even perform an additional test and append the following illegal statements to our code, where we attempt to access the private property `distanceRun` or even apply a value through the `distance` member, which does not have a getter.

```
console.log(myCar.distanceRun);
myCar.distance = 100;
```

Right after inserting these code statements in the playground text field, a red underline will remark that we are attempting to do something that is not correct. Nevertheless, we can carry on and transpile and run the code, since ES5 will honor these practices. All in all, if we attempt to run the `tsc` compiler on this file, the runtime will exit with the following error

trace:

example_26.ts(21,7): error TS1056: Accessors are only available when targeting ECMAScript 5 and higher.
example_26.ts(29,13): error TS2341: Property 'distanceRun' is private and only accessible within class 'Car'.

Interfaces in TypeScript

As applications scale and more classes and constructs are created, we need to find ways to ensure consistency and rules compliance in our code. One of the best ways to address the consistency and type validation issue is to create interfaces.

In a nutshell, an interface is a code blueprint defining a certain fields schema and any types (either classes, function signatures) implementing these interfaces are meant to comply with this schema. This becomes quite useful when we want to enforce strict typing on classes generated by factories, when we define function signatures to ensure that a certain typed property is found in the payload, or other situations.

Let's get down to business! Here, we define the `Vehicle` interface. `Vehicle` is not a class but a contractual schema that any class which implements it must comply with:

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

Any class implementing the `Vehicle` interface must feature a member named `make`, which must be typed as a `string` according to this example. Otherwise, the TypeScript compiler will complain:

```
class Car implements Vehicle {
 // Compiler will raise a warning if 'make' is not defined
    make: string;
}
```

Interfaces are therefore extremely useful to define the minimum set of members any type must fulfill, becoming an invaluable method for ensuring consistency throughout our codebase.

Note

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 for enforcing the existence of certain fields and methods in classes and properties in objects used later on as function parameters, function types, types contained in specific arrays, and even variables. An interface may contain optional members as well and even members typed as other interfaces.

Let's create an example. To do so, we will prefix all our interface types with an `I` (uppercase). This way, it will be easier to find its type when referencing them with our IDE code autocompletion functionality.

First, we define an `IException` interface that models a type with a mandatory message property member and an optional ID number member:

```
interface IException {
```

```
    message: string;
    id?: number;
}
```

We can define interfaces for array elements as well. To do so, we must define an interface with a sole member, defining index as either a number or string (for dictionary collections) and then the type we want that array to contain. In this case, we want to create an interface for arrays containing `IException` types. This is a type comprising a string message property and an optional ID number member, as we said in the previous example:

```
interface IExceptionArrayItem {
    [index: number]: IException;
}
```

Now we define the blueprint for our future class, with a typed array and a method with its returning type defined as well:

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

We can also define interfaces for standalone object types. This is quite useful when it comes to defining a templated constructor or method signatures, which we will see later in this example:

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

Last but not least, in the following class we will implement all these interface types:

```
class ErrorHandler implements IErrorHandler {
    exceptions: IExceptionArrayItem[];
    logAllExceptions: boolean;

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

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

Basically, we are defining an error handler class here that will manage an internal array of exceptions and expose a method to log new exceptions by saving them into the aforementioned array. These two elements are defined by the `IErrorHandler` interface and are mandatory. The class constructor expects the parameters defined by the `IExceptionHandlerSettings` interface and uses them to populate exception member with items typed as `IException`. Instancing the `ErrorHandler` class without the `logAllExceptions`

parameter in the payload will trigger an error.

Let's wrap up this section about interfaces by highlighting that classes can implement more than one interface.

Extending classes with class inheritance

Just like a class can be defined by an interface, it can also extend the members and functionality of other class as if they were its own. We can make a class inherit from another by appending the keyword extends 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;
    }
}
```

Here, we extend from a parent class, Car, which already exposed a make member. We can populate the members already defined by the parent class and even execute their own constructor by executing 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 will still be able to execute the original parent's class methods as it will be still accessible from the super object. Coming back to the interface, they can also inherit definition from other interfaces. Simply put, an interface can inherit from an other interface.

Note

As a word of caution, ES6 and TypeScript do not provide support for multiple inheritance. So, you may want to use composition or middleman classes instead, in case you want to borrow functionalities from different sources.

Decorators in TypeScript

Decorators are a very cool functionality, originally proposed by Google in **AtScript** (a superset of TypeScript that finally got merged into TypeScript back in early 2015) and also a part of the current standard proposition for ECMAScript 7. In a nutshell, decorators are a way to add metadata to class declarations for use by dependency injection or compilation directives (http://blogs.msdn.com/b/somasegar/archive/2015/03/05/typescript-lt-3-angular.aspx). By creating decorators, we are defining special annotations that may have an impact on the way our classes, methods, or functions behave or just simply altering the data we define in fields or parameters. In that sense, decorators are a powerful way to augment our type's native functionalities without creating subclasses or inheriting from other types.

This is, by far, one of the most interesting features of TypeScript. In fact, it is extensively used in Angular 2 when designing directives and components or managing dependency injection, as we will see from Chapter 4, *Enhancing our Components with Pipes and Directives*, onwards.

Note

Decorators can be easily recognized by the @ prefix to their name, and they are usually located as standalone statements above the element they *decorate*, including a method payload or not.

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

Let's take a look at each of them!

Class decorators

Class decorators allow us to augment a class or perform operations over any of its members, and the decorator statement is executed before the class gets instanced.

Creating a class decorator just requires defining a plain function, whose signature is a pointer to the constructor belonging to the class we want to decorate, typed as `Function` (or any other type that inherits from `Function`). The formal declaration defines a `ClassDecorator` as follows:

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

Yes, it is really difficult to grasp what this gibberish means, right? Let's put everything in context through a simple example, like this:

```
function Greeter(target: Function): void {
    target.prototype.greet = function(): void {
        console.log('Hello!');
    }
}

@Greeter
class Greeting {
    constructor() {
        // Implementation goes here...
    }
}

var myGreeting = new Greeting();
myGreeting.greet();    // console will output 'Hello!'
```

As we can see, we have gained a `greet()` method that was not originally defined in the `Greeting` class just by properly decorating it with the `Greeter` decorator.

Extending the class decorator function signature

Sometimes, we might need to customize the way our decorator operates upon instancing it. No worries! We can design our decorators with custom signatures and then have them returning a function with the same signature we define when designing class decorators with no parameters. As a rule of thumb, decorators taking parameters just require a function whose signature matches the parameters we want to configure. Such a function must return another function, whose signature matches that of the decorator we want to define.

The following piece of code illustrates the same functionality as the previous example, but allows developers to customize the greeting message:

```
function Greeter(greeting: string) {
    return function(target: Function) {
        target.prototype.greet = function(): void {
```

```
            console.log(greeting);
        }
    }
}

@Greeter('Howdy!')
class Greeting {
    constructor() {
        // Implementation goes here...
    }
}

var myGreeting = new Greeting();
myGreeting.greet();     // console will output 'Howdy!'
```

Property decorators

Property decorators are meant to be applied on class fields and can be easily defined by creating a `PropertyDecorator` function, whose signature takes two parameters:

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

Possible use cases for this specific type of decorator may encompass from logging the value assigned to class fields when instancing objects of such a class and even reacting to data changes on such fields. Let's see an actual example that encompasses both these behaviors:

```
function LogChanges(target: Object, key: string) {
    var propertyValue: string = this[key];

    if (delete this[key]) {
        Object.defineProperty(target, key, {
          get: function() {
              return propertyValue;
          },
          set: function(newValue) {
              propertyValue = newValue;
              console.log(`${key} is now ${propertyValue}`);
          }
        });
    }
}

class Fruit {
    @LogChanges
    name: string;
}

var fruit = new Fruit();
fruit.name = 'banana';       // console outputs 'name is now banana'
fruit.name = 'plantain'; // console outputs 'name is now plantain'
```

The same logic for parametrized class decorators applies here, although the signature of the returned function is slightly different in order to match that of the parameter-less decorator declaration we already saw.

The following example depicts how we can log changes on a given class property and trigger a custom function when this occurs:

```
function LogChanges(callbackObject: any): Function {
    return function(target: Object, key: string): void {
        var propertyValue: string = this[key];
        if (delete this[key]) {
            Object.defineProperty(target, key, {
              get: function() {
                  return propertyValue;
              },
```

```
            set: function(newValue) {
                propertyValue = newValue;
                callbackObject.onchange.call(this,
                                        propertyValue);
            }
        });
    }
}

class Fruit {
    @LogChanges({
        onchange: function(newValue: string): void {
            console.log(`The fruit is ${newValue} now`);
        }
    })
    name: string;
}

var fruit = new Fruit();
fruit.name = 'banana';     // console: 'The fruit is banana now'
fruit.name = 'plantain'; // console: 'The fruit is plantain now'
```

Method decorators

These special decorators can detect, log, and intervene in how methods are executed. To do so, we just need to define a `MethodDecorator` function whose payload takes the following parameters:

- `target`: This is typed as an object and represents the method being decorated.
- `key`: This is a string that gives the actual name of the method being decorated.
- `value`: This is a property descriptor of the given method. In fact, it's a hash object containing, among other things, a property named value with a reference to the method itself.

Let's see how we can leverage the `MethodDecorator` function in an actual example. Suppose we want to build a multipurpose, signature-agnostic logger that will keep track of the output returned by each method in our class upon execution, including some additional data such as the timestamp when the method was executed:

```
function LogOutput(target: Function, key: string, descriptor: any) {
    var originalMethod = descriptor.value;
    var newMethod = function(...args: any[]): any {
        var result: any = originalMethod.apply(this, args);
        if(!this.loggedOutput) {
            this.loggedOutput = new Array<any>();
        }
        this.loggedOutput.push({
            method: key,
            parameters: args,
            output: result,
            timestamp: new Date()
        });
        return result;
    };

    descriptor.value = newMethod;
}
```

As we mentioned earlier, the descriptor parameter contains a reference to the method we want to decorate. With this in mind, nothing prevents us from replacing such a method by our own. We can take advantage of this newly created method to execute the former by passing along to it the same parameters.

Note

Remember that decorator functions are scoped within the class represented in the target parameter, so we can take advantage of that for augmenting the class with our own custom members. Be careful when doing this, since this might override the already existing members though. For the sake of this example, we won't apply any due diligence over this, but handle this with care in your code in the future.

As we can see, our code is pretty simple and straightforward. We are basically storing a reference to the original method in the `originalMethod` variable, borrowing it from `descriptor.value`. Then, we are building a new method with a `rest` argument that ensures we cover any possible method signature out there. Internally, this new method executes the previous one and stores the result alongside some other data (such as the name of the method executed, the parameters used, and the time and date it occurred on) in a newly created class array field, which is created on the spot in case it didn't exist yet. We then return the computed result, if any, as we would expect from the original method we have just overridden.

Let's put our Shiny decorator to the test! Let's create a class with a method performing method computations and logging and see what happens:

```
class Calculator {
    @LogOutput
    double (num: number): number {
        return num * 2;
    }
}

var calc = new Calculator();
calc.double(11);

console.log(calc.loggedOutput); // Check [Object] array in console
```

Here, our Shiny `Calculator` class exposes a method that will double up any number we pick as an input. But, is it properly logging the operations made? Let's inspect in the value of `calc.loggedOutput` in the browser dev tools console and see what happens.

We can even run the extra mile and add a different method with a completely different type and signature and see whether everything keeps working fine. Add this to the body of the `Calculator` class:

```
    @LogOutput
    doNothing (input: any): any {
        return input;
    }
```

Now, execute the following in the DevTools console command line:

```
calc.doNothing(['Learning Angular 2', 2016]);
```

If you check the value of `calc.loggedOutput`, you will see the new object along with the previous computation we made showing up at the console.

The following line in our example must have caught your attention:

```
this.loggedOutput = new Array<any>();
```

We need to properly type the member we aim to create in our class. To do so, and since we cannot annotate the type the normal way here, we leverage the generic constructor of the

`Array` object passing the any type along between the angle brackets.

This chapter will not cover Generics since they are kind of out of the scope of an introductory book like this, but we encourage you to refer to the official site (http://www.typescriptlang.org/Handbook#generics) and delve deeper into the topic. For the time being, you just need to know that generics allow us to enforce a certain type when executing certain custom methods or using specific classes such as arrays. We will see strictly typed arrays or newly created empty objects that are meant to enforce a certain interface thanks to the functionalities provided by the use of generics. Again, please refer to the official site of the TypeScript language for further information.

Parameter decorators

Our last round of decorators will cover the `ParameterDecorator` function, which taps into parameters located in function signatures. This sort of decorator is not intended to alter the parameter information or the function behavior, but to look into the parameter value and then perform operations elsewhere, such as, for argument's sake, logging or replicating data. The `ParameterDecorator` function takes the following parameters:

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

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

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

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

// The console will output right after the class above is defined:
// 'The parameter in position 0 at Greeter has been decorated'
```

You have probably noticed the weird assignation of the `functionLogged` variable. This is because the value of the `target` parameter will vary depending on the function whose parameters are being decorated. Therefore, it is different if we decorate a `constructor` parameter or a `method` parameter. The former will return a reference to the class prototype and the latter will just return the `constructor` function. The same applies for the `key` parameter, which will be undefined when decorating the `constructor` parameters.

As we mentioned in the beginning of this section, parameter decorators are not meant to modify the value of the parameters decorated or alter the behavior of the methods or constructors where these parameters live. Their purpose is usually to log or *prepare* the container object for implementing additional layers of abstraction or functionality through higher-level decorators, such as method or class decorators. Usual case scenarios for this encompass logging component behavior or managing dependency injection, as we will see in [Chapter 4](), *Enhancing Our Components with Pipes and Directives*.

Organizing our applications with modules

As our applications scale and grow in size, there will be a time when we will need to better organize our code to make it sustainable and more reusable. Modules are the response for this need, so let's take a look at how they work and how we can implement them in our application. Modules can be either internal or external. In this book, we will mostly focus on external modules, but it is a good idea to overview the two types now.

Internal modules

In a nutshell, internal modules are singleton wrappers containing a range of classes, functions, objects, or variables that are scoped internally, away from the global or outer scope. We can publicly expose the contents of a module by prefixing the keyword `export` to the element we want to be accessible from the outside, like this:

```
module Greetings {

    export class Greeting {
        constructor(public name: string) {
            console.log(`Hello ${name}`);
        }
    }

    export class XmasGreeting {
        constructor(public name: string) {
            console.log(`Merry Xmas ${name}`);
        }
    }
}
```

Our `Greetings` module contains two classes that will be accessible from outside the module by importing the module and accessing the class we want to use by its name:

```
import XmasGreeting = Greetings.XmasGreeting;
var xmasGreeting = new XmasGreeting('Joe');
// console outputs 'Merry Xmas Joe'
```

After looking at the preceding code, we can conclude that internal modules are a good way to group and encapsulate elements in a namespace context. We can even split our modules into several files, as long as the module declaration keeps the same name across these files. In order to do so, we will want to reference the different files where we have scattered objects belonging to this module with reference tags:

```
/// <reference path="greetings/XmasGreeting.ts" />
```

The major drawback of internal modules though is that in order to put them to work outside the domain of our IDE, we need to have all of them in the same file or application scope. We can include all the generated JavaScript files as script inserts in our web pages, leverage task runners such as Grunt or Gulp for that, or even use the `--outFile` flag in the TypeScript compiler to have all the `.ts` files found in your workspace compiled into a single bundle using a bootstrap file with reference tags to all the other modules as the starting point for our compilation:

tsc --outFile app.js module.ts

This will compile all the TypeScript files following the trail of dependent files referenced with reference tags. If we forget to reference any file this way, it will not be included in the final build file, so another option is to enlist all the files containing standalone modules in the

compiling command or just add a `.txt` file containing a comprehensive list of the modules to bundle. Alternatively, we can just use external modules instead.

External modules

External modules are pretty much the solution we need when it comes to building applications designed to grow. Basically, each external module works at a file level, where each file is the module itself and the module name will match the filename without the `.js` extension. We do not use the module keyword anymore and each member marked with the `export` prefix will become part of the external module API. The internal module depicted in the previous example would turn into this once conveniently saved in the `Greetings.ts` file:

```
export class Greeting {
    constructor(public name: string) {
        console.log(`Hello ${name}`);
    }
}

export class XmasGreeting {
    constructor(public name: string) {
        console.log(`Merry Xmas ${name}`);
    }
}
```

Importing this module and using its exported classes would require the following code:

```
import greetings = require('Greetings');
var XmasGreetings = greetings.XmasGreetings();
var xmasGreetings = new XmasGreetings('Pete');
// console outputs 'Merry Xmas Pete'
```

Obviously, the `require` function is not supported by traditional JavaScript, so we need to instruct the compiler about how we want that functionality to be implemented in our target JavaScript files. Fortunately, the TypeScript compiler includes the `--module` parameter in its API, so we can configure the dependency loader of choice for our project: `commonjs` for node-style imports, `amd` for RequireJS-based imports, `umd` for a loader implementing the Universal Module Definition specification, or `system` for SystemJS-based imports. We will focus on the SystemJS module loader throughout this book:

tsc --outFile app.js --module commonjs

The resulting file will be properly shimmed, so modules can load dependencies across files using our module loader of choice.

The road ahead

We reached now the end of this chapter. However, we barely scratched the surface of TypeScript. Its huge potential goes way beyond the scope of this shallow introduction and there is definitely more of this which is worth being explored and analyzed in detail. Unfortunately, that is out of the scope of this book and such information is not even required to learn and understand Angular 2, although this knowledge will obviously expand the boundaries of your code practice. We would suggest you to check the following aspects of the language specification as a starting point:

- Generics
- Mixins
- Intersection types
- Union types
- Tuples
- Creating declaration type files

If you want to delve deeper into TypeScript, I would recommend you read *TypeScript Essentials* by *Christopher Nance*, *Learning TypeScript* by *Remo H. Jansen*, or the more advanced *Mastering TypeScript* by *Nathan Rozentals*, all by *Packt Publishing*.

Summary

This was definitely a long read, but this introduction to TypeScript was absolutely necessary in order to understand the logic behind many of the most brilliant parts of Angular 2. It gave us the chance to not only introduce the language syntax, but also explain the rationale behind its success as the syntax of choice for building the Angular 2 framework. We reviewed its type architecture and how we can create advanced business logic designing functions with a wide range of alternatives for parametrized signatures and even discovered how to bypass the issues related with scope by using the powerful new arrow functions. Probably the most relevant part of this chapter encompassed the 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 major features explored in this chapter and, as we will see very soon, having a sound knowledge of these mechanisms is paramount to understand how dependency injection works in Angular 2.

With all this knowledge at our disposal, we can now resume our investigation of Angular 2 and confront with confidence the relevant parts of component creation such as style encapsulation, output formatting, and so on. Chapter 3, *Implementing Properties and Events in Our Components*, will expose us to advanced template creation techniques, data-binding techniques, directives, and pipes. All these features will allow us to put in practice all this newly gained knowledge of TypeScript.

Chapter 3. Implementing Properties and Events in Our Components

So far, we have had the opportunity to take a bird's eye overview of what a component is in the new Angular ecosystem, what is its role, how it behaves, and what tools are required to start building our own components to represent widgets and pieces of functionality. In addition, TypeScript turns out to be the perfect companion for this endeavor, so we seem to have everything that we need to further explore the possibilities that Angular 2 brings to the game with regards to creating interactive components that expose properties and emit events.

In this chapter, we will:

- Discover all the syntactic possibilities at our disposal to bind content in our templates
- Create public APIs for our components so that we can benefit from their properties and event handlers
- See how to implement data binding in Angular 2
- Reduce the complexity of CSS management with view encapsulation

A better template syntax

In [Chapter 1](#), *Creating Our Very First Component in Angular 2,* we saw how to embed HTML templates in our components, but we didn't even scratch the surface of template development for Angular 2. As we will see later in this book, template implementation is tightly coupled with the principles of Shadow DOM design and brings forth a lot of syntactic sugar to ease the task of binding properties and events in our views in a declarative fashion.

In a nutshell, Angular components may expose a public API that allows them to communicate with other components or containers. This API may encompass input properties, which we use to feed the component with data. It also may expose output properties we can bind event listeners to, thereby getting prompt information about changes in the state of the component.

Let's take a look at the way Angular solves the problem of injecting data in and out of our components through quick and easy examples. Please focus on the philosophy behind these properties. We will have a chance to see them in action later on when we follow up with our pomodoro project.

Data bindings with input properties

Let's revisit the pomodoro functionality that we already saw in , *Creating Our Very First Component in Angular 2*, and let's imagine that we want our component to have a configurable attribute so that we can increase or decrease the countdown time:

```
<pomodoro-timer [seconds]="25"></pomodoro-timer>
```

Please note the attribute wrapped between brackets. This informs Angular that this is an input property. The class that models the `pomodoro-timer` component will contain a setter function for the `seconds` property, which will react to changes in that value by updating its own countdown duration. We can inject a data variable or an actual hardcoded value, in which case we will have to wrap it around single quotes within the double quotes should such a value be a text string.

Sometimes, we will see this syntax while injecting data into our component's custom properties, while at other times, we will use this very bracket syntax to make native HTML attributes reactive to component fields, like this:

```
<h1 [hidden]="hideMe">
  This text will not be visible if 'hideMe' is true
</h1>
```

Some extra syntactic sugar when binding expressions

The Angular team has made available some shortcuts for performing common transformations in our component directives and DOM elements, such as tweaking attributes and class names or applying styles. Here, we have some examples of great time-savers when declaratively defining bindings in our properties:

```
<div [attr.hidden]="isHidden">...</div>
<input [class.is-valid]="isValid">
<div [style.width.px]="myWidth">...</div>
```

In the first case, `div` will enable the `hidden` attribute should the `isHidden` expression evaluate to `true`. Besides Boolean values, we can bind any other data type, such as a string value. In the second case, the `is-valid` class name will be injected in the `class` attribute if the `isValid` expression evaluates to `true`. In our third example, `div` will feature a `style` attribute that shows off a `width` property meant to be set with the value of the `myWidth` expressions in pixels. You can find more examples of this syntactic sugar in the Angular 2 cheat sheet (https://angular.io/cheatsheet) available at the official Angular site.

Event binding with output properties

Let's imagine we want our pomodoro timer component to notify us when the countdown is finished so that we can perform some other actions outside the realm of the component. We can achieve such functionality with an output property like this:

```
<pomodoro-timer (countdownComplete)="onCountownCompleted()"></pomodoro-timer>
```

Note the attribute wrapped between braces. This informs Angular that such an attribute is, in fact, an output property that will trigger the event handler we bind to it. In this case, we will want to create an `onCountownCompleted` event handler on the container object that wraps this component.

Tip

By the way, the camel case is not a coincidence. It is a naming convention applied to all output and input property names in Angular 2.

We will find output properties mapped to interaction events that we already know, such as `click`, `mouseover`, `mouseout`, `focus`, and more.

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

Input and output properties in action

The best way to grasp the concepts detailed in the earlier sections is by practice. Let's strip down the pomodoro timer example that we saw in [Chapter 1](#), *Creating Our Very First Component in Angular 2*, and discuss a simpler example. Open the `pomodoro-timer.ts` file and replace its contents with the following component class:

```
import { Component } from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'
})
class CountdownComponent {
  seconds: number = 25;
  intervalId: number;

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearInterval(this.intervalId);
    }
  }
}
```

Great! We have just defined a simple but highly effective countdown timer component that will count down to 0 from 25 seconds. (Do you see the `seconds` field up there? TypeScript supports the initialization of members upon declaring them). A simple `setInterval` loop executes a custom private function named `tick()` that decreases the value of seconds until it reaches zero, in which case we just clear the interval.

Please notice that we did not include any call to the `bootstrap` function to instantiate this component in a HTML document, even though we are effectively importing the function. This is actually pretty common, since we only call the `bootstrap` function to instantiate the top parent component (also known as `root` component) and all the other components, defined as child components of the former, will be automatically instantiated in cascade.

However, now we just need to embed this component somewhere, so let's create another component with no functionality other than acting as a HTML wrapper host for the previous component. Create this new component right after the `CountdownComponent` class in its same file:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: '<countdown></countdown>'
```

```
})
class PomodoroTimerComponent {}

bootstrap(PomodoroTimerComponent);
```

As we can see there, we introduced a new property named `directives` in the component initialization and declared `CountdownComponent` there. Components in Angular 2 are basically directives with a view template. We can also find directives with no view, which basically add new functionalities to their host element; or they just act as custom elements without a UI that wraps other elements. Alternatively, they simply provide further functionalities to other components by means of their API.

We will explore directives in detail in the next chapter and also along the book. For now, let's just point out that when we define a component containing other components, as we have just done, we will have to explicitly declare the immediate children components' classes in the `directives` array parameter. You must be wondering why have we created this host or parent `PomodoroTimerComponent` component with no implementation. Soon, we will flesh it out with some more features, but for now let's use it as a proof of concept for how to initiate a component tree.

Setting up custom values declaratively

You will probably agree on the fact that having the functionality of setting up custom countdown timers would be nice, right? Input properties turn out to be an excellent way to achieve this. In order to leverage this functionality, we will have to tweak the import statement at the top of the file:

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

Let's update our pomodoro timer accordingly:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'

})
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  // Rest of implementation remains the same...
}
```

You might have already noticed that we are no longer initializing the `seconds` field, and it is now decorated with a property decorator (as we saw in Chapter 2, *Introducing TypeScript*). We have just started to define the API of our component.

Note

Property naming is case sensitive, and the convention enforced by Angular 2 is to apply

camel case to component input and, as we will see shortly, output properties alike.

Next, we just need to add the desired property in our container component's template:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
               <img src="assets/img/pomodoro.png" />
               <countdown [seconds]="25"></countdown>
             </div>`

})
```

Please note that we have not updated the `PomodorotimerComponent` at all. We only updated its `CountdownComponent` children component. However, its brand new API becomes available to any component that eventually includes it in its own template as a child component, so we can setup its properties declaratively right from the template, or even bind a value imperatively from a property located at the `PomodorotimerComponent` controller class if we wish.

Tip

When flagging a class property with `@Input()`, we can configure the name we want this property to have upon instantiating the component in the HTML. To do so, we just need to introduce our name of choice in the decorator signature, like this `@Input('name_of_the_property')`. In any event, this practice is discouraged since exposing property names in the component API distinct from the ones defined in its controller class can only lead to confusion.

Communicating between components through custom events

Now that our child component is being configured by its parent component, how can we achieve communication from the child to the parent? This is where custom events come to the rescue! In order to create proper event bindings, we just need to configure an output property in our component and attach an event handler function to it.

In order to trigger custom events, we will need to bring `EventEmitter` to the party, along with the `@Output` decorator, whose functionality is exactly the opposite to what we learned regarding the `@Input` decorator:

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

`EventEmitter` is the built-in event bus of Angular 2. In a nutshell, the `EventEmitter` class provides support for emitting Observable data and subscribing Observer consumers to data changes. Its simple interface, which basically encompass two methods, `emit()` and `subscribe()`, can therefore be used to trigger custom events and listen to events as well, both synchronously or asynchronously. We will discuss Observables in more in detail in [Chapter 6](#), *Asynchronous Data Services with Angular 2*. For the time being, we can get away with the

idea that we will be using the `EventEmitter` API to spawn events that listener methods in the components hosting our event-emitting component can observe and attach event handlers to. These events acquire visibility outside the scope of the component through any of its properties annotated with the `@Input()` decorator.

The following code shows an actual implementation that follows up from the previous example:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>'
})
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  @Output() complete: EventEmitter<any> = new EventEmitter();

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearTimeout(this.intervalId);
      // An event is emitted upon finishing the countdown
      this.complete.emit(null);
    }
  }
}
```

A new property named `complete` is conveniently annotated with the `EventEmitter` type and initialized on the spot. Later on we will access its `emit` method to spawn a custom event as soon as the countdown ends. The `emit()` method needs one mandatory parameter of any type, so we can send a data value to the event subscribers (or `null` if not required).

Now, we just need to set up our host component so that it will listen to this `complete` event or `output` property and subscribe an event handler to it:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
             <img src="assets/img/pomodoro.png" />
             <countdown [seconds]="25"
               (complete)="onCountdownCompleted()">
             </countdown>
           </div>`
})
class PomodoroTimerComponent {
  onCountdownCompleted(): void {
    alert('Time up!');
  }
}
```

Tip

Why complete and not onComplete?

Angular 2 provides support for an alternative syntax named *canonical form* for both input and output properties. In the case of input properties, a property represented as `[seconds]` could be represented as `bind-seconds`, without the need for brackets. With regards to output properties, these can be represented as `on-complete` instead of `(complete)`. That is why we never prefix output property names with an on prefix, since that would concur on output properties such as `on-on-complete` in case we eventually decide to favor the canonical syntax form in our projects.

Emitting data through custom events

Now that we know how to emit custom events from our component API, why don't we take a step further and send data signals beyond the scope of the component? We already discussed that the emit() event of the EventEmitter<T> class accepts in its signature any given data of the type represented by the T annotation. Let's extend our example to notify the progress of the countdown. Why would we ever want to do this? Basically, our component displays on screen a visual countdown, but we might want to watch the countdown progress programmatically in order to take action once the countdown is finished or reaches a certain point.

Let's update our timer component with another output property that matches the original and emits a custom event on each iteration of the seconds property, as follows:

```
class CountdownComponent {
  @Input() seconds: number;
  intervalId: number;
  @Output() complete: EventEmitter<any> = new EventEmitter();
  @Output() progress: EventEmitter<number> = new EventEmitter();

  constructor() {
    this.intervalId = setInterval(() => this.tick(), 1000);
  }

  private tick(): void {
    if (--this.seconds < 1) {
      clearTimeout(this.intervalId);
      this.complete.emit(null);
    }
    this.progress.emit(this.seconds);
  }
}
```

Now, let's rebuild our host component's template to reflect the actual progress of the countdown. We already do so by displaying the countdown, but that is a feature handled internally by the CountdownComponent. Now, we will keep track of the countdown outside this component:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  template: `<div class="container text-center">
             <img src="assets/img/pomodoro.png" />
             <countdown [seconds]="25"
               (progress)="timeout = $event"
               (complete)="onCountdownCompleted()">
             </countdown>
             <p *ngIf="timeout < 10">
               Beware! Only <strong>{{timeout}} seconds</strong> left.
             </p>
           </div>`
})
class PomodoroTimerComponent {
```

```
  timeout: number;
  onCountdownCompleted(): void {
    alert('Time up!');
  }
}
```

We took advantage of this round of changes to formalize the `timeout` value as a property of the host component. This allows us to bind new values to that property in our custom event handlers, as we did in the preceding example. Rather than binding an event handler method to the `(progress)` handler, we refer to the `$event` reserved variable. It is a pointer to the payload of the `progress` output property that reflects the value we pass to the `emit()` function when executing `this.progress.emit(this.seconds)`. In short, `$event` is the value assumed by `this.seconds` inside `CountdownComponent`. By assigning such value to the `timeout` class property within the template, we are also updating the binding expressed in the paragraph we just inserted in the template. This paragraph will only become visible when `timeout` is lower than 10.

```
<countdown [seconds]="25"
  on-progress="timeout = $event"
  on-complete="onCountdownCompleted()">
</countdown>
```

Local references in templates

We have previously seen how we can bind data to our templates using data interpolation with the double curly braces syntax. Besides this, we will quite often spot named identifiers prefixed by a hash symbol (#) in the elements belonging to our components or even regular HTML controls. These reference identifiers, namely local names, are used to refer to the components flagged with them in our template views and then access them programmatically. They can also be used by components to refer to other elements in the virtual DOM and access its properties.

In the previous section, we saw how we could subscribe to the countdown progress through the `progress` event. But, what if we could inspect the component in depth, or at least its public properties and methods, and read the value that the `seconds` property takes on each tick interval without having to listen to the `progress` event? Well, setting a local reference on the component itself will open the door to its public façade.

Let's flag the instance of our `CountdownComponent` in the `PomodoroTimerComponent` template with a local reference named `#counter`. From that very moment, we will be able to directly access the component's public properties, such as `seconds`, and even bind it in other locations of the template. This way, we do not even need to rely on the `progress` event emitter or the `timeout` class field, and we can even manipulate the value of such properties. This is shown in the following code:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  encapsulation: ViewEncapsulation.None,
  template: `<div class="container text-center">
            <img src="assets/img/pomodoro.png" />
            <countdown [seconds]="25"
              (complete)="onCountdownCompleted()"
              #counter>
            </countdown>
            <p>
              <button class="btn btn-default" (click)="counter.seconds =
25">Reset countdown to 25 seconds
              </button>
            </p>
            <p *ngIf="counter.seconds < 10">
              Beware! Only
              <strong>{{counter.seconds}} seconds</strong>
              left.
            </p>
          </div>`
})
class PomodoroTimerComponent {
  // timeout: number; /* No longer required */
  onCountdownCompleted(): void {
    alert('Time up!');
  }
```

}

Alternative syntax for input and output properties

Besides the `@Input()` and `@Output()` decorators, there is an alternative syntax where we can define input and output properties in our components by means of the `@Component` decorator. Its metadata implementation provides support for both features through the `inputs` and `outputs` property names, respectively.

The `CountdownComponent` API could therefore be implemented like this:

```
@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>',
  inputs: ['seconds'],
  outputs: ['complete', 'progress']
})
class CountdownComponent {
  seconds: number;
  intervalId: number;
  complete: EventEmitter<any> = new EventEmitter();
  progress: EventEmitter<number> = new EventEmitter();

  // Etc...
}
```

All in all, this syntax is discouraged and has been included here for reference purposes only. In the first place, we duplicate code by defining the names of our API endpoints in two places at the same time, increasing the risk of errors when refactoring code. It is also a common convention to keep the decorator implementations as lean as possible in order to improve readability.

Tip

I strongly suggest that you stick to the `@Input` and `@Output` decorators.

Configuring our template from our component class

The `Component` metadata also supports several settings that contribute to easy template management and configuration. On the other hand, Angular 2 takes advantage of the CSS encapsulation functionalities of Web Components.

Internal and external templates

As our applications grow in size and complexity, chances are that our templates will grow as well, hosting other components and bigger chunks of HTML code. Embedding all this code in our component class definitions will become a cumbersome and unpleasant task and quite prone to errors by the way. In order to prevent this from happening, we can leverage the `templateUrl` property, pointing to a standalone HTML file that contains our component HTML markup.

Back to our previous example, we can refactor the `@Component` decorator of our `PomodoroTimerComponent` class to point to an external html file containing our template. Create a new file named `pomodoro-timer.html` in the same workspace where our `pomodoro-timer.ts` file lives and populate it with the same HTML we configured in our `PomodoroTimerComponent` class:

```
<div class="container text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro"/>
  <countdown
    [seconds]="25"
    (complete)="onCountdownCompleted()"
    #counter>
  </countdown>
  <p>
    <button class="btn btn-default"(click)="counter.seconds = 25">
      Reset countdown to 25 seconds
    </button>
  </p>
  <p *ngIf="counter.seconds < 10">
    Beware! Only
    <strong>{{ counter.seconds }} seconds</strong>
    left.
  </p>
</div>
```

Now, we can polish our `@Component` decorator to point to that file instead of defining the HTML inside the decorator metadata:

```
@Component({
  selector: 'pomodoro-timer',
  directives: [CountdownComponent],
  templateUrl: './pomodoro-tasks.html'
})
class PomodoroTimerComponent {
  // Class follows below...
}
```

Note

External templates follow a certain convention in Angular 2, enforced by the most popular Angular 2 coding style guides out there, which is to share the same filename than the component they belong to, including any filename prefix or suffix we might append to the

component filename. We will see this when exploring component naming conventions in [Chapter 5](), *Building an Application with Angular 2 Components*. This way, it is easier to recognize, or even search with your IDE search built-in fuzzy finder tool, what HTML file is in fact the template of a specific component.

What is the threshold for creating standalone templates rather than keeping the template markup inside the component? It depends on the complexity and size of the template. Common sense will be your best advisor in this case.

Encapsulating CSS styling

In order to better encapsulate our code and make it more reusable, we can define CSS styling within our components. These internal style sheets are a good way to make our components more shareable and maintainable. There are three different ways of defining CSS styling for our components.

The styles property

We can define styles for our HTML elements and class names through the style property in the component decorator, like this:

```
@Component({
  selector: 'my-component',
  styles: [`
    p {
      text-align: center;
    }
    table {
      margin: auto;
    }`]
})
```

This property will take an array of strings containing CSS rules each and apply them to the template markup by embedding those rules at the head of the document as soon as we bootstrap our application. We can either inline the styling rules in a single line, or take advantage of ES2015 template strings to indent the code and make it more readable as depicted in the example above.

The styleUrls property

Just like `styles`, `styleUrls` will accept an array of strings, although each one will represent a link to an external style sheet though. This property can be used alongside the `styles` property as well, defining different sets of rules where required:

```
@Component({
  selector: 'my-component',
  styleUrls: ['path/to/my-stylesheet.css'],
  styles: [`
    p {
      text-align: center;
    }
    table {
      margin: auto;
    }`]
})
```

Inline style sheets

We can also attach the styling rules to the template itself, no matter whether it's an inline template or a template served through the `templateUrl` parameter:

```
@Component({
  selector: 'app',
  template: `
    <style> p { color: red; } </style>
    <p>I am a red paragraph</p>`
})
```

Managing view encapsulation

All the preceding sections (`styles`, `styleUrls`, and inline style sheets) will be governed by the usual rules of CSS specificity (https://developer.mozilla.org/en/docs/Web/CSS/Specificity). CSS management and specificity becomes a breeze on browsers that support Shadow DOM, thanks to scoped styling. CSS styles apply to the elements contained in the component but do not spread beyond its boundaries.

On top of that, Angular will embed these style sheets at the head of the document, so they might affect other elements of our application. In order to prevent this from happening, we can set up different levels of view encapsulation.

In a nutshell, encapsulation is the way Angular needs to manage CSS scoping within the component for both Shadow DOM-compliant browsers and those that do not support it. For all this, we leverage the `ViewEncapsulation` enum, which can take any of these values:

- **Emulated**: This is the default option, and it basically entails an emulation of native scoping in Shadow DOM through sandboxing the CSS rules under a specific selector that points to our component. This option is preferred to ensure that our component styles will not be affected by other existing libraries on our site.
- **Native**: Use the native Shadow DOM encapsulation mechanism of the renderer, and it only works on browsers that support Shadow DOM.
- **None**: Template or style encapsulation is not provided. The styles will be injected as is into the document's header.

Let's check out an actual example. First, import the `ViewEncapsulation` enum into the script, and then create an `encapsulation` property with the `Emulated` value. Then, let's create a style rule for our countdown text so any `<h1>` (!) tag is rendered in dark red:

```
import {
  Component,
  EventEmitter,
  Input,
  Output,
  ViewEncapsulation
} from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';

@Component({
  selector: 'countdown',
  template: '<h1>Time left: {{seconds}}</h1>',
  styles: ['h1 { color: #900 }'],
  encapsulation: ViewEncapsulation.Emulated
})
class CountdownComponent {
  // Etc...
}
```

Now, click on the browser's dev tools inspector and check the generated HTML to discover

how Angular 2 injected the CSS inside the page `<HEAD>` block. The just injected style sheet has been sandboxed to ensure that the global CSS rule we defined at the component setup in a very non-specific way for all `<h1>` elements only applies to matching elements scoped by the `CountdownComponent` component exclusively.

We recommend that you try out different values and see how the CSS code is injected into the document. You will immediately notice the different grades of isolation that each variation provides.

Summary

This chapter guided us through the options available in Angular 2 for creating powerful APIs for our components, so we can provide high levels of interoperability between components, configuring its 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 own template, setting the ground up for larger component trees in our applications. Output parameters give the layer of interactivity we need by turning our components into event emitters so they can properly communicate in an agnostic fashion with any parent component that might eventually host them. Template references paved the way to create references in our custom elements that we can use as accessors to their properties and methods from within the template in a declarative fashion. We also discussed how we could isolate the component's HTML template in an external file in order to ease its future maintainability and how to do the same with any style sheet we wanted to bind to the component, in case we do not want to bundle the component styles inline. An overview of the built-in features for handling view encapsulation in Angular 2 gave us some additional insights on how we can benefit from Shadow DOM's CSS scoping on a per component basis and how we can polyfill it when not supported.

We still have much more to learn regarding template management in Angular 2, mostly with regards to the two concepts that you will use extensively along your journey with Angular. I am referring to Directives and Pipes, which we will cover extensively in Chapter 4, *Enhancing our Components with Pipes and Directives*.

Chapter 4. Enhancing Our Components with Pipes and Directives

In the previous chapters, we built several components that rendered data on screen with the help of input and output properties. We will leverage the knowledge in this chapter to take our components to the next level with the use of directives and pipes. In a nutshell, while pipes give us the opportunity to digest and transform the information we bind in our templates, directives allow us to conduct more ambitious functionalities where we can access the host element properties and also bind our very own custom event listeners and data bindings.

In this chapter, we will:

- Have a comprehensive overview of the built-in directives of Angular 2
- Discover how we can refine our data output with pipes
- See how we can design and build our own custom pipes and directives
- Leverage built-in objects for manipulating our templates
- Put all the preceding topics and many more into practice by following up on our pomodoro project to build a fully interactive to-do items table

Directives in Angular 2

Angular 2 defines directives as components without views. In fact, a component is a directive with an associated template view. This distinction is used because directives are a prominent part of the Angular 2 core and each (plain directives and component directives) needs the other to exist. Directives can basically affect the way HTML elements or custom elements behave and display their content.

Core directives

Let's take a closer look at the framework's core directives, and then you will learn how to build your own directives later on in this chapter.

NgIf

As the official documentation states, the `NgIf` directive removes or recreates a portion of the DOM tree based on an expression. If the expression assigned to the `NgIf` directive evaluates to false, then the element is removed from the DOM. Otherwise, a clone of the element is reinserted into the DOM. We could enhance our countdown timer by leveraging this directive, like this:

```
<pomodoro-timer [seconds]="timeout"></pomodoro-timer>
<p *ngIf="timeout === 0">Time up!</p>
```

When our pomodoro timer reaches `0`, the paragraph that displays the `Time up!` text will be rendered on the screen. You have probably noticed that asterisk that prepends the directive. This is because Angular embeds the HTML control marked with the `NgIf` directive (and all its HTML subtrees, if any) in a `<template>` tag, which will be used later on to render the content on the screen. Covering how Angular treats templates is definitely out of the scope of this book, but let's just point out that this is syntactic sugar provided by Angular to act as a shortcut to that other, more verbose syntax based on template tags.

Perhaps you are wondering what difference does it make to render some chunk of HTML on screen with `*ngIf="conditional"` rather than with `[hidden]="conditional"`. The former will clone and inject pieces of templated HTML snippets in the markup, removing it from the DOM when the condition evaluates to false, while the latter does not inject or remove any markup from the DOM. It simply sets the visibility of the already existing chunk of HTML annotated with that DOM attribute.

NgFor

The `NgFor` directive allows us to iterate through a collection (or any other iterable object) and bind each of its items to a template of our choice, where we can define convenient placeholders to interpolate the item data. Each instantiated template is scoped to the outer context, where the loop directive is placed, so we can access other bindings. Let's imagine we have a component named `Staff`: it features a field named `employees`, which represents an array of `Employee` objects. We can enlist those `employees` and job titles in this way:

```
<ul>
  <li *ngFor="let employee of employees; let i = index; let last = last">
    Employee #{{i}}: - {{employee.name}}, {{employee.position}}
    <span *ngIf="last"><br />End of list</span>
  </li>
</ul>
```

As we can see in the example provided, we turn each item fetched from the iterable object on

each loop into a local reference so that we can easily bind this item in our template. Here, we also use the syntax sugar that we used in the previous section, and Angular gives us the opportunity to assign `index` to a scoped variable that will be set to the current loop iteration in the template context. We can also assign `last` to a scoped variable that will inform whether the item is the last one in the iteration.

This directive observes changes in the underlying iterable object and will add, remove, or sort the rendered templates as items are added, removed, or reordered in the collection.

NgStyle

As you probably have guessed already, this directive allows us to bind CSS styles by evaluating a custom object or expression. We can bind an object whose keys and values map CSS properties, or just define specific properties and bind data to them:

```
<p [ngStyle]="{ 'color': myColor, 'font-weight': myFontWeight }">I am red and
bold</p>
```

If our component defines the `myColor` and `myFontWeight` properties with the `red` and `bold` values, respectively, the color and weight of the text will change accordingly. The directive will always reflect the changes made within the component, and we can also pass an object instead of binding data on a per property basis:

```
<p [ngStyle]="myCssConfig">I am red and bold</p>
```

NgClass

Similar to `NgStyle`, `NgClass` allows us to define and toggle class names programmatically in a DOM element using a convenient declarative syntax. This syntax has its own intricacies, however. Let's see each one of the three case scenarios available for this example:

```
<p [ng-class]="{{myClassNames}}">Hello Angular!</p>
```

For instance, we can use a string type so that if `myClassNames` contains a string with one or several classes delimited by a space, all of them will be bound to the paragraph.

We can use an array as well so that each element will be added.

Last but not least, we can use an object in which each key corresponds to a CSS class name referred to by a Boolean value. Each key name marked as true will become an active class. Otherwise, it will be removed. This is usually the preferred way of handling class names.

NgSwitch, NgSwitchWhen, and NgSwitchDefault

The `NgSwitch` directive is used to switch templates within a specific set depending on the condition required for displaying each one. The implementation follows several steps, therefore three different directives are explained in this section.

NgSwitch will evaluate a given expression and then toggle and display those child elements marked with an `ngSwitchWhen` attribute directive, whose value matches the value thrown by the expression defined in the parent `ngSwitch` element. A special mention is required about the children element marked with the `ngSwitchDefault` directive attribute. This attribute qualifies the template that will be displayed when no other value defined by its `ngSwitchWhen` siblings matches the parent conditional expression.

We'll see all of this in an example:

```
<div [ngSwitch]="weatherForecastDay">
    <template ngSwitchWhen="today">{{weatherToday}}</template>
    <template ngSwitchWhen="tomorrow">
      {{weatherTomorrow}}</template>
    <template ngSwitchDefault>
        Pick a day to see the weather forecast
    </template>
</div>
```

The parent `[ngSwitch]` parameter evaluates the `weatherForecastDay` context variable, and each nested `ngSwitchWhen` directive will be tested against it. We can use expressions instead, but we want to wrap `ngSwitchWhen` in brackets so that Angular can properly evaluate its content as context variables instead of taking it as a text string.

Note

At the time of closing the writing of this book, the Angular core team is discussing the convenience of renaming `ngSwitchWhen` to `ngSwitchCase` in order to keep consistent with JavaScript and TypeScript switch/case keywords, and also with other i18n directives such as `NgPlural` and `NgPluralCase`. It is quite likely that this breaking change will make it to Angular 2 final so please refer to the online documentation to double check the final syntax for `NgSwitch` template cases.

Coverage for the `NgPlural` and `NgPluralCase` sits outside of the scope of this book, but basically provide a convenient way to render or remove templates DOM blocks that match a switch expression, either strictly numeric or just a string, in a similar fashion to how the `NgSwitch` and `NgSwitchWhen` directives do.

Manipulating template bindings with Pipes

So, we saw how we can use directives to render content depending on the data that our component classes manage, but there is another powerful feature that we will be using thoroughly in our daily practice with Angular. We are talking about *Pipes*.

Pipes allow us to filter and funnel the outcome of our expressions on a view level to transform or just better display the data we are binding. Their syntax is pretty simple, basically consisting of the pipe name following the expression that we want to transform, separated by a pipe symbol (hence the name):

```
@Component({
  selector: "greeting",
  template: "HELLO {{ name | uppercase }}"
})
class GreetingComponent {
  name: string;
}
```

In the preceding example, we are displaying an uppercase greeting on the screen. Since we do not know whether the name will be in uppercase or not, we ensure a consistent output by transforming the value of the name whenever it is not an uppercase version at the view level. Pipes are chainable, and Angular has a wide range of pipe types already baked in. As we will see further in this chapter, we can also build our own pipes to fine-grain data output in cases where the built-in pipes are simply not enough.

The uppercase/lowercase pipe

The name uppercase/lowercase pipe says it all. As in the example provided previously, this pipe sets the string output in uppercase or lowercase. Insert the following code anywhere in your view and check out the output for yourself:

```
<p>{{ 'hello world' | uppercase }}</p>
<!-- outputs 'HELLO WORLD' -->

<p>{{ 'wEIrD hElLo' | lowercase }}</p>
<!-- output is 'weird hello' -->
```

The number, percent, and currency pipes

Numeric data can come in a wide range of flavors, and this pipe is especially convenient when it comes to better formatting and localizing the output. These pipes use the Internationalization API, and therefore they are reliable in Chrome and Opera browsers only.

The number pipe

The number pipe will help us define the grouping and sizing of numbers using the active locale in our browser. Its format is as follows:

```
expression | number[:digitInfo]
```

Here, `expression` is a number and `digitInfo` has the following format:

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

Each binding would correspond to the following:

- `minIntegerDigits`: The minimum number of integer digits to use. It defaults to 1.
- `minFractionDigits`: The minimum number of digits after the fraction. It defaults to 0.
- `maxFractionDigits`: The maximum number of digits after the fraction. It defaults to 3.

Note

Keep in mind that the acceptable range for each of these numbers and other details will depend on your native internationalization implementation.

The percent pipe

The percent pipe formats a number as local percent. Other than this, it inherits from the Number pipe so that we can further format the output to provide a better integer and decimal sizing and grouping. Its syntax is as follows:

```
expression | percent[:digitInfo]
```

The currency pipe

Formats a number as a local currency, providing support for selecting the currency code such as `USD` for the US dollar or `EUR` for the euro and setting up how we want the currency info to be displayed. Its syntax is as follows:

```
expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]
```

In the preceding statement, `currencyCode` is obviously the ISO 4217 currency code, while `symbolDisplay` is a Boolean that indicates whether to use the currency symbol (for example, $) or the currency code (for, example `USD`) in the output. The default for this value is `false`. Similar to the number and percent pipes, we can format the output to provide a better integer and decimal sizing and grouping through the `digitInfo` value:

```
<p>{{ 11256.569 | currency:"GBP":true:'4.1-2' }}</p>
<!-- output is '£11,256.57' -->
```

The slice pipe

The purpose of this pipe is equivalent to the role played by `Array.prototype.slice()` and `String.prototype.slice()` when it comes to subtracting a subset (`slice`) of a collection list, array, or string, respectively. Its syntax is pretty straightforward and follows the same conventions as those of the aforementioned `slice()` methods:

```
expression | slice:start[:end]
```

Basically, we configure a starting index where we will begin slicing either the items array or the string on an optional end index, which will fall back to the last index on the input when omitted.

Note

Both start and end arguments can take positive and negative values, as the JavaScript `slice()` methods do. Refer to the JavaScript API documentation for a full rundown on all the available scenarios.

Last but not least, please note that when operating on a collection, the returned list is always a copy—even when all elements are being returned.

The date pipe

You must have already guessed that the Date pipe formats a date value as a string based on the requested format. The time zone of the formatted output will be the local system time zone of the end user's machine. Its syntax is pretty simple:

```
expression | date[:format]
```

The `expression` input must be a date object or a number (milliseconds since the UTC epoch). The format argument is highly customizable and accepts a wide range of variations based on date-time symbols. For our convenience, some aliases have been made available as shortcuts to the most common date formats:

- `'medium'`: This is equivalent to `'yMMMdjms'` (for example, Sep 3, 2010, 12:05:08 PM for en-US)
- `'short'`: This is equivalent to `'yMdjm'` (for example, 9/3/2010, 12:05 PM for en-US)
- `'fullDate'`: This is equivalent to `'yMMMMEEEEd'` (for example, Friday, September 3, 2010 for en-US)
- `'longDate'`: This is equivalent to `'yMMMMd'` (for example, September 3, 2010)
- `'mediumDate'`: This is equivalent to `'yMMMd'` (for example, Sep 3, 2010 for en-US)
- `'shortDate'`: This is equivalent to `'yMd'` (for example, 9/3/2010 for en-US)
- `'mediumTime'`: This is equivalent to `'jms'` (for example, 12:05:08 PM for en-US)
- `'shortTime'`: This is equivalent to `'jm'` (for example, 12:05 PM for en-US)

The JSON pipe

JSON is probably the most straightforward pipe in its definition; it basically takes an object as an input and outputs it in JSON format:

```
{{ { name: 'Eve', age: 43 } | json }}
```

Here is the output:

```
{ "name": "Eve", "age": 43 }
```

The replace pipe

The replace pipe operates pretty much like the `String.prototype.replace()` function of the JavaScript API, and it will evaluate a string expression, or a number that will be treated as a string either way, against a given pattern. All matches found will be then replaced by a given string replacement. We can also introduce a function, or a reference to a function, that will receive the match string found. The overall syntax is as follows:

```
expression | replace:pattern:replacement
```

It is important to note that the pattern can be configured as a regular expression. In fact, Angular 2 uses regular expressions under the hood to find string matches so make sure you escape any special character like parentheses, brackets, and so on.

The i18n pipes

As part of Angular's strong commitment to providing a strong internationalization toolset, a reduced set of pipes targeting common i18n use cases have been made available. This book will only cover the two major ones, but it is quite likely that more pipes will be released in the future. Please refer to the official documentation for further information after finishing this chapter.

The i18nPlural pipe

The i18nPlural pipe has a simple usage, where we just evaluate a numeric value against an object mapping different string values to be returned depending on the result of the evaluation. This way, we can render different strings on our template depending if the numeric value is zero, one, two, more than N, and so on. Talking about pomodoros, we could slide this in our template:

```
<h1> {{ pomodoros | i18nPlural:pomodorosWarningMapping }} </h1>
```

Then, we can have this mapping as a field of our component controller class:

```
class MyPomodorosComponent {
  pomodoros: number;
  pomodorosWarningMapping: any = {
    '=0': 'No pomodoros for today',
    '=1': 'One pomodoro pending',
    'other': '# pomodoros pending'
  }
}
```

We even bind the numeric value evaluated in the expression by introducing the '#' placeholder in the string mappings. When no matching value is found, the pipe will fall back to the mapping set with the key `'other'`.

The i18nSelect pipe

The i18nSelect pipe is similar to `i18nPlural` pipe, but evaluates a string value instead. This pipe is perfect for localizing text interpolations or providing distinct labels depending on state changes, for instance. For example, we could recap on our pomodoro timer and serve the UI in different languages:

```
<button (click)="togglePause()">
  {{ languageCode | i18nSelect: localizedLabelsMap }}
</button>
```

In our controller class, we can populate `localizedLabelsMap` as follows:

```
class PomodoroTimerComponent {
  languageCode: string = 'fr';
  localizedLabelsMap: any = {
    'en': 'Start timer',
```

```
      'es': 'Comenzar temporizador',
      'fr': 'Démarrer une séquence',
      'other': 'Start timer'
    }
    ...
}
```
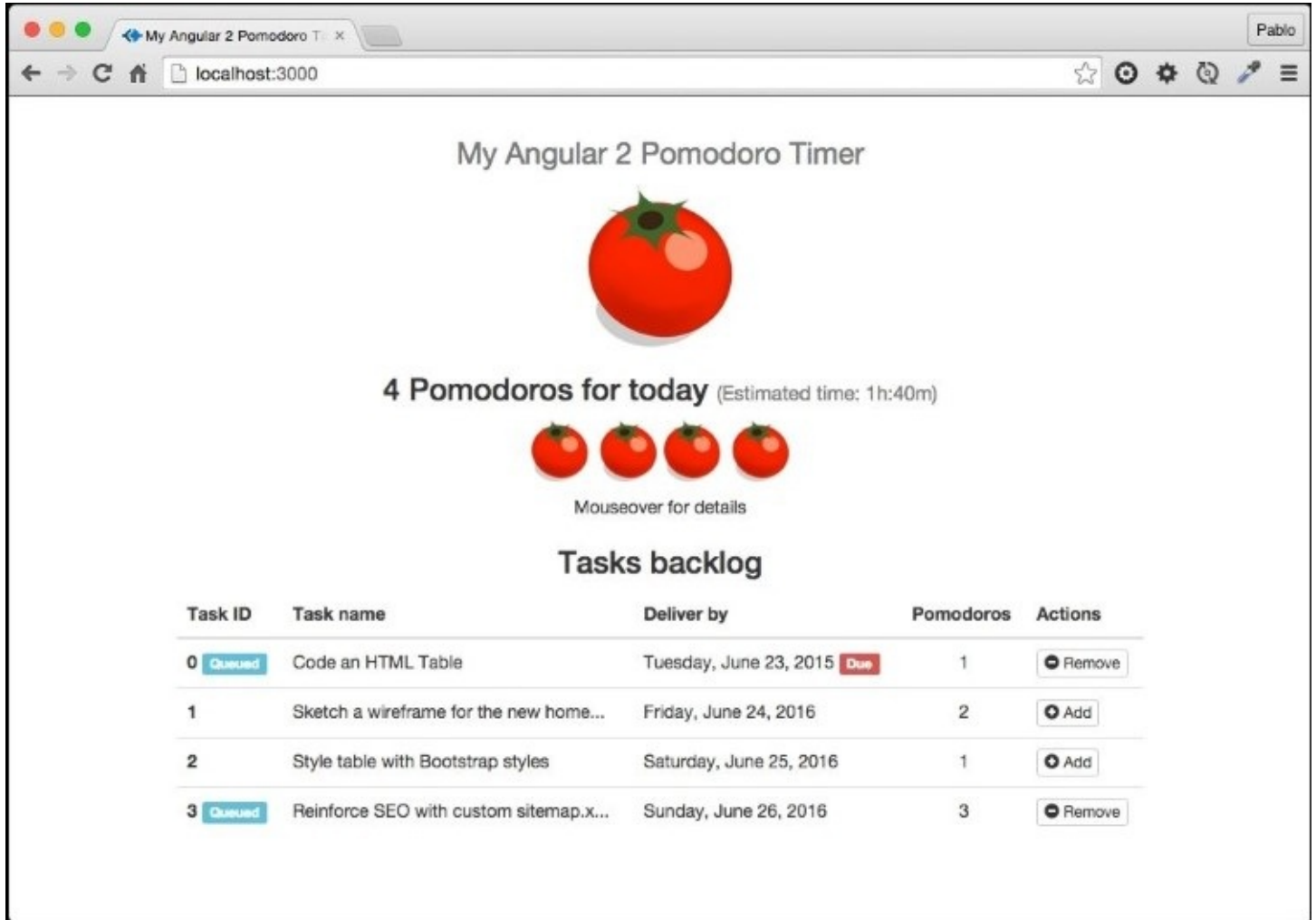
It is important to note that we can put this convenient pipe to use in use cases other than localising components, but to provide string bindings depending on map keys and the like. Same as the `i18nPlural` pipe, when no matching value is found, the pipe will fall back to the mapping set with the key `'other'`.

The async pipe

Sometimes, we manage observable data or only data that is handled asynchronously by our component class, and we need to ensure that our views promptly reflect the changes in the information once the observable field changes or asynchronous loading has been accomplished after the view has been rendered. The async pipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the async pipe marks the component to be checked for changes.

Putting it all together in the Pomodoro task list

Now that you have learned all the elements that allow you to build full-blown components, it's time to put all of this fresh knowledge into practice. In the next pages we are going to build a simple task list manager for our pomodoro application. In it, we will see a tasks table containing the to-do items we need to achieve:

We will also queue up tasks straight from the backlog of tasks available. This will help showing the time required to accomplish all the queued tasks and see how many pomodoros are defined in our working agenda.

Setting up our main HTML container

Before building the actual component we need to set up our work environment first and in order to do so we will reuse the same HTML boilerplate file we used in the previous component. Please set aside the work you've done so far and keep the `package.json`, `tsconfig.json`, `typings.json` and `index.html` files we used in previous examples. Feel free to reinstall the modules required in case you need to, and replace the contents of the body tag in our `index.html` template:

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro Tasks</strong>
    </div>
  </div>
</nav>
<pomodoro-tasks></pomodoro-tasks>
```

In a nutshell, we have just updated the title of the header layout above our new `<pomodoro-tasks>` custom elements, which replaces the previous `<pomodoro-timer>`. You might want to update the configuration of the `System.import()` command to point to our new compiled component class:

```
System.import('built/pomodoro-tasks')
      .then(null, console.error.bind(console));
```

Building our task list table with Angular directives

Create an empty `pomodoro-tasks.ts` file. You might want to use this newly created file to build our new component from scratch and embed on it the definitions of all the accompanying pipes, directives, and components we will see later in this chapter.

Note

Real-life projects are never implemented this way, since our code must conform to the "one class, one file" principle, taking advantage of ECMAScript modules for gluing things together. [Chapter 5](), *Building an Application with Angular 2 Components* will introduce you to a common set of good practices for building Angular 2 applications, including strategies for organizing your directory tree and your different elements (components, directives, pipes, services, and so on) in a sustainable way. This chapter, on the contrary, will leverage `pomodoro-tasks.ts` to include all the code in a central location and then provide a bird's eye view of all the topics we will cover now without having to go switching across files. Bear in mind that this is in fact an anti-pattern, but for instructional purposes we will take this approach in this chapter for the last time. The order in which elements are declared within the file is important. Refer to the code repository in GitHub if exceptions rise.

Before moving on with our component, we need to import the dependencies required, formalize the data model we will use to populate the table, and then scaffold some data that will be served by a convenient service class.

Let's begin by adding to our `pomodoro-tasks.ts` file the following code block, importing all the tokens we will require in this chapter. Pay special attention to the tokens we are importing from the Angular 2 library. We have covered `Component` and `Input` already, but all the rest will be explained later in this chapter:

```
import {
  Component,
  Input,
  Pipe,
  PipeTransform,
  Directive,
  OnInit,
  HostListener
} from '@angular/core';
import { bootstrap } from '@angular/platform-browser-dynamic';
```

With the dependency tokens already imported, let's define the data model for our tasks, next to the block of imports:

```
/// Model interface
interface Task {
  name: string;
  deadline: Date;
  queued: boolean;
  pomodorosRequired: number;
```

```
}
```

The schema of a `Task` model interface is pretty self-explanatory. Each task has a name, a deadline, a field informing how many pomodoros need to be shipped, and a Boolean field named `queued` that defines if that task has been tagged to be done in our next pomodoro session.

Tip

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

Now, we need some data and a service wrapper class to deliver such data in the form of a collection of `Task` objects. The `TaskService` class defined here will do the trick, so append it to your code right after the `Task` interface:

```
/// Local Data Service
class TaskService {
  public taskStore: Array<Task> = [];

  constructor() {
    const tasks = [
      {
        name: "Code an HTML Table",
        deadline: "Jun 23 2015",
        pomodorosRequired: 1
      }, {
        name: "Sketch a wireframe for the new homepage",
        deadline: "Jun 24 2016",
        pomodorosRequired: 2
      }, {
        name: "Style table with Bootstrap styles",
        deadline: "Jun 25 2016",
        pomodorosRequired: 1
      }, {
        name: "Reinforce SEO with custom sitemap.xml",
        deadline: "Jun 26 2016",
        pomodorosRequired: 3
      }
    ];

    this.taskStore = tasks.map(task => {
      return {
        name: task.name,
        deadline: new Date(task.deadline),
        queued: false,
        pomodorosRequired: task.pomodorosRequired
      };
    });
  }
```

```
}
```

This data store is pretty self-explanatory: it exposes a `taskStore` property returning an array of objects conforming to the `Task` interface (hence benefiting from static typing) with information about the name, deadline, and time estimate in pomodoros.

Now that we have a data store and a model class, we can begin building an Angular component which will consume this data source to render the tasks in our template view. Insert the following component implementation after the code you wrote before:

```
/// Component classes

/// - Main Parent Component

@Component({
  selector: 'pomodoro-tasks',
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class TasksComponent {
  today: Date;
  tasks: Task[];

  constructor() {
    const TasksService: TasksService = new TasksService();
    this.tasks = taskService.taskStore;
    this.today = new Date();
  }
};

bootstrap(TasksComponent);
```

As you can see, we have defined and instantiated through the `bootstrap` function a new component named `TasksComponent` with the selector `<pomodoro-tasks>` (we already included it when we were populating the main `index.html` file, remember?). This class exposes two properties: today's date and a tasks collection that will be rendered in a table contained in the component's view, as we will see shortly. To do so, it instantiates in its constructor the data source that we created previously, mapping it to the array of models typed as `Task` objects represented by the `tasks` field. We also initialize the `today` property with an instance of the JavaScript built-in `Date` object, which contains the current date.

Tip

As you have seen, the component selector does not match its controller class naming. We will delve deeper into naming conventions at the end of this chapter, as a preparation for *Chapter 5, Building an Application with Angular 2 Components*.

Let's create the stylesheet file now, whose implementation will be really simple and straightforward. Create a new file named `pomodoro-tasks.css` at the same location where our component file lives. You can then populate it with the following styles ruleset:

```
h3, p {
   text-align: center;
}
table {
  margin: auto;
  max-width: 760px;
}
```

This newly created stylesheet is so simple that it might seem a bit too much to have it as a standalone file. However, this comes as a good opportunity to showcase in our example the functionalities of the `styleUrls` property of the component metadata.

Things are quite different in regards of our HTML template. This time we will not hardcode our HTML template in the component either, but we will point to an external HTML file to better manage our presentation code. Please create an HTML file and save it as `pomodoro-tasks.html` in the same location where our main component's controller class exists. Once it is created, fill it in with the following HTML snippet:

```
<div class="container text-center">
  <img src="assets/img/pomodoro.png" alt="Pomodoro"/>
  <div class="container">
    <h4>Tasks backlog</h4>
    <table class="table">
      <thead>
        <tr>
          <th>Task ID</th>
          <th>Task name</th>
          <th>Deliver by</th>
          <th>Pomodoros</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let task of tasks; let i = index">
          <th scope="row">{{i}}</th>
          <td>{{task.name | slice: 0:35 }}
            <span [hidden]="task.name.length < 35">...</span>
          </td>
          <td>{{task.deadline | date: 'fullDate' }}
            <span *ngIf="task.deadline < today" class="label label-danger">
Due
            </span>
          </td>
          <td class="text-center">{{task.pomodorosRequired}}</td>
          <td>
            [Future options...]
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

We are basically creating a table that features a neat styling based on the Bootstrap

framework. Then, we render all our tasks using the always convenient `NgFor` directive, extracting and displaying the `index` of each item in our collection as we explained while overviewing the `NgFor` directive earlier in this chapter.

Please look at how we formatted the output of our task's name and deadline interpolations by means of pipes, and how conveniently we display (or not) an ellipsis to indicate when the text exceeds the maximum number of characters we allocated for the name by turning the HTML `hidden` property into a property bound to an Angular expression. All this presentation logic is topped with a red label, indicating whether the given task is due whenever its end date is prior to this day. If you execute the preceding code, this page will show up on the screen:
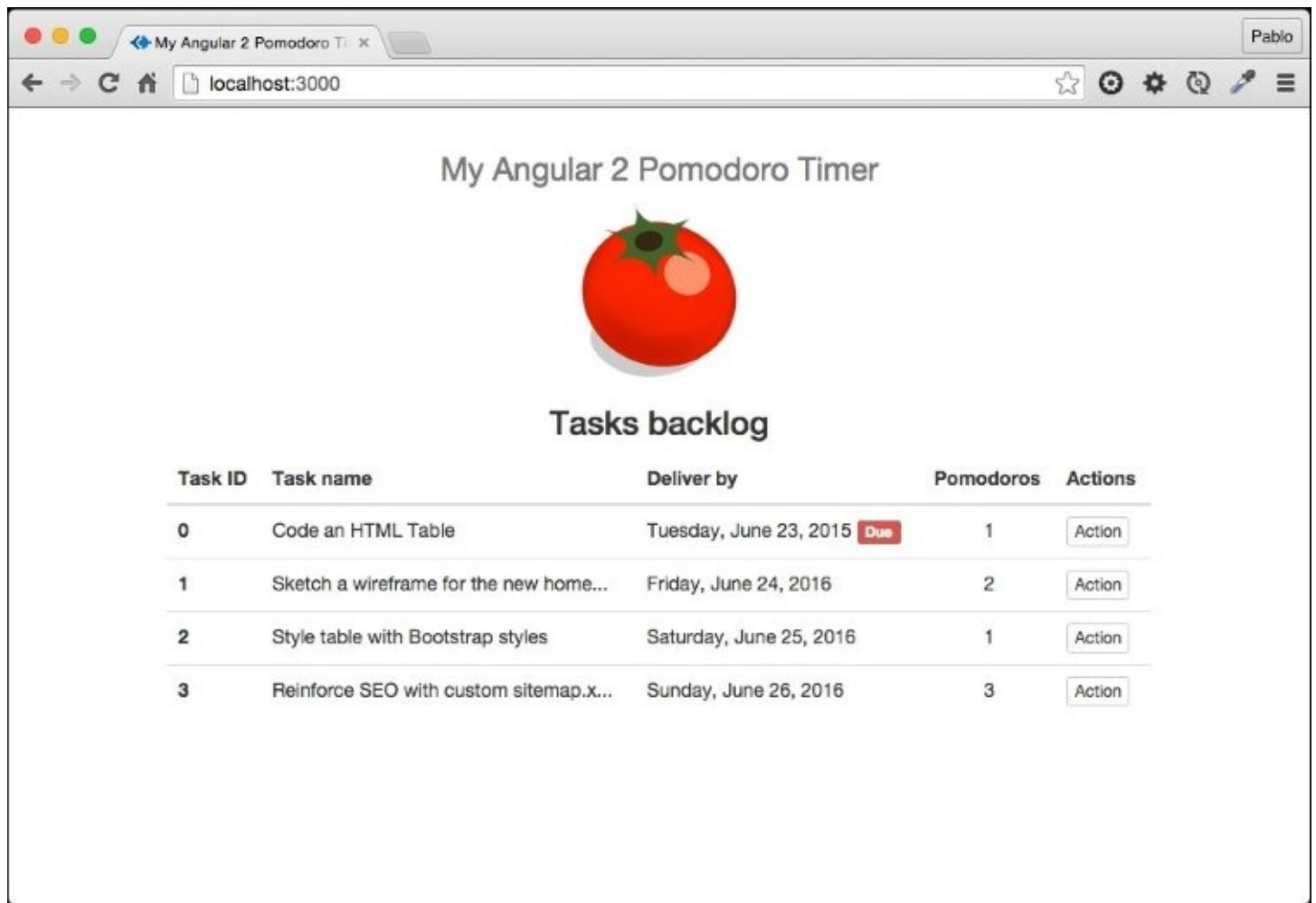
You have probably noticed that those action buttons do not exist in our current implementation. We will fix this in the next section, playing around with state in our components. Back in Chapter 1, *Creating Our Very First Component in Angular 2*, we touched upon the click event handler for stopping and resuming the pomodoro countdown, and then delved deeper into the subject in Chapter 3, *Implementing Properties and Events in Our Components*, where we covered output properties. Let's continue on our research and see how we can hook up DOM event handlers with our component's public methods, adding a rich layer of interactivity to our components.

Toggling tasks in our task list

Add the following method to your `TasksComponent` controller class. Its functionality is pretty basic; we just literally toggle the value of the `queued` property for a given `Task` object instance:

```
toggleTask(task: Task): void {
  task.queued = !task.queued;
}
```

Now, we just need to hook it up with our view buttons. Update our view to include a click attribute (wrapped in braces so that it acts as an output property) in the button created within the `NgFor` loop. Now that we will have different states in our `Task` objects, let's reflect this in the button labels by implementing a `NgSwitch` structure all together:

```
<table class="table">
  <thead>
    <tr>
      <th>Task ID</th>
      <th>Task name</th>
      <th>Deliver by</th>
      <th>Pomodoros</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#task of tasks; #i = index">
      <th scope="row">{{i}}
        <span *ngIf="task.queued" class="label label-info">
          Queued
        </span>
      </th>
      <td>{{task.name | slice: 0:35 }}
        <span [hidden]="task.name.length < 35">...</span>
      </td>
      <td>{{task.deadline | date: 'fullDate' }}
        <span *ngIf="task.deadline < today" class="label label-danger">
          Due
        </span>
      </td>
      <td class="text-center">{{task.pomodorosRequired}}</td>
      <td>
        <button
          type="button"
          class="btn btn-default btn-xs"
          (click)="toggleTask(task)"
          [ngSwitch]="task.queued">
          <template [ngSwitchWhen]="false">
            <i class="glyphicon glyphicon-plus-sign"></i>
            Add
          </template>
          <template [ngSwitchWhen]="true">
            <i class="glyphicon glyphicon-minus-sign"></i>
            Remove
```

```
          </template>
          <template ngSwitchDefault>
            <i class="glyphicon glyphicon-plus-sign"></i>
            Add
          </template>
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

Our brand new button can execute the `toggleTask` method in our component class, passing as an argument the `Task` object that corresponds to that iteration of `NgFor`. On the other hand, the preceding `NgSwitch` implementation allows us to display different button labels and icons depending on the state of the `Task` object at any given time.

Tip

We are decorating the newly created buttons with font icons fetched from the Glyphicons font family. The icons are part of the Bootstrap CSS bundle we installed previously and are in no means related to Angular 2. Feel free to skip its use or to replace it by another icon font family.

Execute the code as it is now and check out the results yourself. Neat, isn't it? But maybe we can get more juice from Angular 2 by adding more functionality to the task list.

Displaying state changes in our templates

Now that we can pick the tasks to be done from the table, it would be great to have some kind of visual hint of how many pomodoro sessions we are meant to achieve. The logic is as follows:

- The user reviews the tasks on the table and picks the ones to be done by clicking on each one.
- Every time a row is clicked, the underlying `Task` object state changes and its Boolean `queued` property is toggled.
- The state change is reflected immediately on the surface by displaying a "queued" label on the related task item.
- The user gets prompt information of the amount of pomodoro sessions required and a time estimation to deliver them all.
- We see how a row of pomodoro icons are displayed above the table, displaying the sum of pomodoros from all the tasks set to be done.

This functionality will have to react to the state changes of the set of `Task` objects we're dealing with. The good news is that thanks to Angular 2's very own change detection system, making components fully aware of state changes is extremely easy.

Thus, our very first task will be to tweak our `TasksComponent` class to include some way to compute and display how many tasks are queued up. We will use that information to render or not a block of markup in our component where we will inform how many pomodoros we have lined up and how much aggregated time it will take to accomplish them all.

The new `queuedPomodoros` field of our class will provide such information, and we will want to insert a new method named `updateQueuedPomodoros()` in our class that will update its numeric value upon instantiating the component or enqueueing tasks. On top of that, we will create a key/value mapping we can use later on to render a more expressive title header depending on the amount of queued pomodoros thanks to the `I18nPlural` pipe:

```
class TasksComponent {
  today: Date;
  tasks: Task[];
  queuedPomodoros: number;
  queueHeaderMapping: any = {
    '=0': 'No pomodoros',
    '=1': 'One pomodoro',
    'other': '# pomodoros'
  };

  constructor() {
    const TasksService: TasksService = new TasksService();
    this.tasks = taskService.taskStore;
    this.today = new Date();
    this.updateQueuedPomodoros();
  }
```

```
  toggleTask(task: Task): void {
    task.queued = !task.queued;
    this.updateQueuedPomodoros();
  }

  private updateQueuedPomodoros(): void {
    this.queuedPomodoros = this.tasks
      .filter((task: Task) => task.queued)
      .reduce((pomodoros: number, queuedTask: Task) => {
      return pomodoros + queuedTask.pomodorosRequired;
    }, 0);
  }
};
```

The `updateQueuedPomodoros()` method makes use of JavaScript's native `Array.filter()` and `Array.reduce()` methods to build a list of queued tasks out of the original `tasks` collection property. The `reduce` method applied over the resulting array gives us the total number of pomodoros required. With a stateful computation of the number of queued pomodoros now available, it's time to update our template accordingly. Go to `pomodoro-tasks.html` and inject the following chunk of HTML right before the `<h4>Tasks backlog</h4>` element. The code is as follows:

```
  <div>
  <h3>
    {{ queuedPomodoros | i18nPlural:queueHeaderMapping }} for today
    <span class="small" *ngIf="queuedPomodoros > 0">(Estimated time: {{
queuedPomodoros * 25 }})</span>
  </h3>
  </div>
  <h4>Tasks backlog</h4>
  <!-- rest of template remains the same -->
```

The preceding block renders an informative header title at all times, even when no pomodoros have been queued up. We also bind that value in the template and use it to estimate through an expression binding the amount of minutes required to go through each and every pomodoro session required.

Tip

We are hardcoding the duration of each pomodoro in the template. Ideally, such constant value should be bound from an application variable or a centralized setting. Don't worry, we will see how we can improve this implementation in the next chapters.

Save your changes and reload the page, and then try to toggle some task items on the table to see how the information changes in real time. Exciting, isn't it?

Embedding child components

Now, let's start building a tiny pomodoro icon component that will be nested inside the `TasksComponent` component. This new component will display a smaller version of our big pomodoro icon, which we will use to display on the template the amount of pomodoros lined up to be done, as we described earlier in this chapter. Let's pave the way towards component trees, which we will analyze in detail in [Chapter 5](), *Building an Application with Angular 2 Components*. For now, just include the following component class before the `TasksComponent` class you built earlier:

Our component will expose a public property named `task` in which we can inject a `Task` object. The component will use this `Task` object binding to replicate the image rendered in the template as many times as pomodoro sessions are required by this task in its `pomodorosRequired` property, all this by means of a `NgFor` directive.

In our `pomodoro-tasks.ts` file, inject the following block of code before our `TasksComponent`:

```
@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
                   src="/assets/img/pomodoro.png"
                   width="50">`
})
class TaskIconsComponent implements OnInit {
  @Input() task: Task;
  icons: Object[] = [];

  ngOnInit() {
    this.icons.length = this.task.pomodorosRequired;
    this.icons.fill({ name: this.task.name });
  }
}
```

Our new `TaskIconsComponent` features a pretty simple implementation, with a very intuitive selector matching its camel-cased class name and a template where we duplicate the given `<img>` tag as many times as objects are populated in the `icons` array property of the controller class, which is populated with the native fill method of the `Array` object in the JavaScript API (the `fill` method fills all the elements of an array with a static value passed as an argument), within `ngOnInit()`. Wait, what is this? Shouldn't we implement the loop populating the `icons` array member in the constructor instead?

This method is one of the lifecycle hooks we will overview in the next chapter, and probably the most important one. The reason why we populate the `icons` array field here and not in the constructor method is because we need each and every data-bound properties to be properly initialized before proceeding to run the `for` loop. Otherwise, it will be too soon to access the input value `task` since it will return an `undefined` value.

Tip

The `OnInit` interface demands an `ngOnInit()` method to be integrated in the controller class that implements such -interface, and it will be executed once all input properties with a binding defined have been checked. We will take a bird's eye overview of component lifecycle hooks in [Chapter 5](#), *Building an Application with Angular 2 Components*.

Still, our new component needs to find its way to its parent component. So, let's insert a reference to the `component` class in the `directives` property of the `TasksComponent` decorator settings:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [TaskIconsComponent],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
```

Do you remember that we mentioned that components are basically directives with custom views? If so, then we will want to use the `directives` property of each component every time we want to nest another component within. This explains the case for using the `directives` property here.

Our next step will be to inject the `<pomodoro-task-icons>` element in the `TasksComponent` template. Go back to `pomodoro-tasks.html` and update the code located inside the conditional block meant to be displayed when `queuedPomodoros` is greater than zero. The code is as follows:

```
<div>
  <h3>
    {{ queuedPomodoros | i18nPlural:queueHeaderMapping }} for today
    <span class="small" *ngIf="queuedPomodoros > 0">(Estimated time: {{
queuedPomodoros * 25 }})</span>
  </h3>
  <p>
    <span *ngFor="let queuedTask of tasks">
      <pomodoro-task-icons
        [task]="queuedTask"
        (mouseover)="tooltip.innerText = queuedTask.name"
        (mouseout)="tooltip.innerText = 'Mouseover for details'">
      </pomodoro-task-icons>
    </span>
  </p>
  <p #tooltip *ngIf="queuedPomodoros > 0">Mouseover for details</p>
</div>
<h4>Tasks backlog</h4>
<!-- rest of template remains the same -->
```

There is still some room for improvement though. Unfortunately, the icon size is hardcoded in the `TaskIconsComponent` template and that makes it harder to reuse that component in other contexts where a different size might be required. Obviously, we could refactor the `TaskIconsComponent` class to expose a `size` input property and then bind the value received straight into the component template in order to resize the image accordingly:

```
@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
                  src="/assets/img/pomodoro.png"
                  width="{{size}}">`
})
class TaskIconsComponent implements OnInit {
  @Input() task: Task;
  icons: Object[] = [];
  @Input() size: number;

  ngOnInit() {
    ...
  }
}
```

Then, we just need to update the implementation of `pomodoro-tasks.html` to declare the value we need for the size:

```
<span *ngFor="let queuedTask of tasks">
  <pomodoro-task-icons
    [task]="queuedTask"
    size="50"
    (mouseover)="tooltip.innerText = queuedTask.name"
    (mouseout)="tooltip.innerText = 'Mouseover for details'">
  </pomodoro-task-icons>
</span>
```

Please note that the size attribute is not wrapped between brackets because we are binding a hardcoded value. If we wanted to bind a component variable, that attribute should be properly declared as `[size]="{{mySizeVariable}}"`.

Let's summarize what we did:

- We inserted a new DOM element that will show up only when we have pomodoros queued up.
- We displayed an actual header telling us how many pomodoros we are meant to achieve, by binding the `queuedPomodoros` property in an H3 DOM element, plus a total estimation in minutes for accomplishing all of this contained in the `{{ queuedPomodoros*25 }}` expression.
- The `NgFor` directive allows us to iterate through the `tasks` array. In each iteration, we render a new `<pomodoro-task-icons>` element.
- We bound the `Task` model object of each iteration, represented by the `queuedTask` reference, in the `task` input property of the `<pomodoro-task-icons>` in the loop template.
- We took advantage of the `<pomodoro-task-icons>` element to include additional mouse event handlers that point to the following paragraph, which has been flagged with the `#tooltip` local reference. So, every time the user hovers the mouse over the pomodoro icon, the text beneath the icons row will display the respective pomodoro's task name.

We ran the extra mile, turning the size of the icon rendered by `<pomodoro-task-icons>` into a

configurable property as part of the component API. We now have pomodoro icons that get updated in real time as we toggle the information on the table. New problems have arisen, however. Firstly, we are displaying pomodoro icon components matching the required pomodoros of each task, without filtering out those which are not queued. On the other hand, the overall estimation of time required to achieve all our queued pomodoros displays the gross number of minutes, and this information will make no sense as we add more and more pomodoros to the working plan.

Perhaps, it's time to amend this. It's a good thing that custom pipes have come to the rescue!

Building our own custom pipes

We have already seen what pipes are and what their purpose is in the overall Angular ecosystem, but now we are going to dive deeper into how we can build our own set of pipes to provide custom transformations to data bindings.

Anatomy of a custom pipe

Pipes are very easy to define. First of all, we need to import the `Pipe` decorator from the Angular core library and create a new class decorated with this decorator. This new class has to be named with our selector of choice in the decorator configuration and implement the `PipeTransform` interface.

The class implementation is pretty simple as well. It just consists of a mandatory method required by the `PipeTransform` interface, named `transform`, which will return a type of our choice (usually the type corresponding to the input that we feed the pipe with) and two parameters. The input itself is the first parameter, followed by an optional spread argument (refer to [Chapter 2](#), *Introducing TypeScript* to look into spread arguments in TypeScript) containing the settings that configure the pipe in our view:

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

@Pipe({
  name: 'myPipeName',
  pure: false // optional, default is true
})
class MyPipe implements PipeTransform {
  transform(value: any, ...args: any[]): any {
    // We apply transformations to the input value here
    return something;
  }
}
@Component({
  selector: 'my-selector',
  pipes: [MyPipe],
  template: '<p>{{ myVariable | myPipeName: "bar" }}</p>'
})
class myComponent {
  myVariable: string = 'Foo';
}
```

Tip

In the preceding example, we created an impure pipe named `MyPipe` that would apply some transformations to its input according to the parameters provided when applying it in the component defined next.

Just as with custom directives, components must explicitly state in the `pipes` property what custom pipes are implementing.

Regarding the optional `pure` property in the `Pipe` decorator, we must clarify that pipes are stateless. This means that an instance of a pipe will be reused and the pipe will be called only when its arguments change. In other words, when the pipe transforms the input, it disregards the original input and focuses on the copy that was just made. If the original input changes later on, the changes will not be reflected in the view. Fortunately, we can enable the state on

pipes through the `pure` Boolean property. When set to `false`, the pipe will keep the state of the values it transforms, and it will parse and transform the underlying expression again as soon as Angular's change detection system checks that the source data has changed.

Let's put this concept to work by creating a couple of custom pipes for our component.

A custom pipe to better format time output

Watching the gross number of minutes summed up when lining up tasks to be done is not very intuitive, so we need a way to deconstruct this value into hours and minutes. Our pipe will have the name `pomodoroFormattedTime` and will be implemented by the `FormattedTimePipe` class, whose unique transform method receives a number representing a total number of minutes and returns a string (proving that pipes do not need to return the same type as they receive in the payload) in a readable time format:

```
@Pipe({
  name: 'pomodoroFormattedTime'
})
class FormattedTimePipe implements PipeTransform {
  transform(totalMinutes: number): string {
    let minutes: number = totalMinutes % 60;
    let hours: number = Math.floor(totalMinutes / 60);
    return `${hours}h:${minutes}m`;
  }
}
```

Tip

We should not skip the opportunity to highlight that the naming convention for Pipes is, same as we saw with Components, the name of the pipe class with the `Pipe` suffix plus a selector matching that name without the suffix. The difference here is that we represent the pipe selector in camel case and prefix it with `pomodoro` for our example. Why this mismatch between the pipe controller's class name and the selector? It is common practice to prefix the selector strings of our custom pipes and directives with a custom prefix in order to prevent collisions with other selectors defined by third party pipes and directives.

Please remember that custom pipes do not become available in our templates automatically; they have to be explicitly declared in the pipes property of the decorator configuration of each component that wants to use them. In our case, it is the `TasksComponent`:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent],
  pipes: [FormattedTimePipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
  // Class implementation remains the same
}
```

Finally, we just need to tweak the HTML in the `pomodoro-tasks.html` template file to ensure that our EDT expression is properly formatted:

```
<span class="small">
  (Estimated time: {{queuedPomodoros*25 | pomodoroFormattedTime}})
```

```
</span>
```

Now reload the page and toggle some tasks. The estimated time will be properly rendered in hours and minutes.

Filtering out data with custom filters

As we noticed already, we are displaying at this moment a pomodoro icon component for each and every task in the collection served from the tasks service, without filtering out what tasks are marked as `queued` and which aren't. Pipes provide a convenient way to map, transform and digest data bindings, so we can leverage its functionalities for filtering out the tasks binding in our `NgFor` loop to return only those tasks that are marked as queued.

The logic will be pretty simple: since the tasks binding is an array of `Task` objects, we just need to make use of the `Array.filter()` method to fetch only those `Task` objects whose queued property is set to true. We might run the extra mile and configure our pipe to take one Boolean argument indicating whether we want to filter out queued or unqueued tasks. The implementation of these requirements is as follows, where you can see again the conventions in place for the selector and class names:

```
@Pipe({
  name: 'pomodoroQueuedOnly',
  pure: false
})
class QueuedOnlyPipe implements PipeTransform {
  transform(task: Task], ...args: any[]): Task[] {
    return tasks.filter((task: Task) => {
      return task.queued === args[0];
    });
  }
}
```

The implementation is pretty straightforward, so we will not get into detail about it here. However, there is something that is worth highlighting at this stage: this is an *impure* pipe. Bear in mind that the tasks binding is a collection of stateful objects that will change in length and content as the user toggles tasks on the table. For that reason, we need to instruct the pipe to take advantage of Angular's change detection system so its output is checked by the latter on every cycle regardless of whether its input has changed or not. Configuring the pure property of the pipe decorator as false will do the trick then.

Now, we just need to update the pipes property of the component using this pipe:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent],
  pipes: [FormattedTimePipe, QueuedOnlyPipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
  // Class implementation remains the same
}
```

Then, update the `NgFor` block in `pomodoro-tasks.html` to properly filter out the unqueued

tasks:

```
<span *ngFor="#queuedTask of tasks | pomodoroQueuedOnly: true">
  <pomodoro-task-icons
    [task]="queuedTask"
    (mouseover)="tooltip.innerText = queuedTask.name"
    (mouseout)="tooltip.innerText = 'Mouseover for details'">
  </pomodoro-task-icons>
</span>
```

Please check how we configured the pipe as `pomodoroQueuedOnly: true`. Replacing the Boolean parameter value by `false` will give us the chance to enlist the pomodoros pertaining to the queues we have not picked.

Save all your work and reload the page, toggling some tasks then. You will see how our overall UI reacts to the latest changes accordingly, and we only enlist the pomodoro icons pertaining to the amount of pomodoros required of queued tasks only.

Building our own custom directives

Custom directives encompass a vast world of possibilities and use cases, and we would need an entire book for showcasing all the intricacies and possibilities they offer.

In a nutshell, directives allow you to attach advanced behaviors to elements in the DOM. If a directive has a template attached, then it becomes a component. In other words, components are Angular directives with a view, but we can build directives with no attached views that will be applied to already existing DOM elements, making its HTML contents and standard behavior immediately accessible to the directive. This applies to Angular components as well, where the directive will just access its template and custom attributes and events when necessary.

Anatomy of a custom directive

Declaring and implementing a custom directive is pretty easy. We just need to import the `Directive` class to provide decorator functionalities to its accompanying controller class:

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

Then we define a controller class annotated by the `Directive` decorator, where we will define the directive selector, input and output properties (if required), optional events applied to the host element, and injectable provider tokens, should our directive's constructor require specific types to be instantiated by the Angular 2 injector when instancing itself (we will cover this in detail in [Chapter 5](#), *Building an Application with Angular 2 Components*):

```
@Directive({
  selector: '[selector]',
  inputs: ['inputPropertyName'],
  outputs: ['outputPropertyName'],
  host: {
    '(event1)': 'onMethod1($event)',
    '(target:event2)': 'onMethod2($event)',
    '[prop]': 'expression',
    'attributeName': 'attributeValue'
  },
  providers: [MyCustomType]
})
class myDirective {
  @Input() otherInputPropertyName: any;
  @Output() otherOutputPropertyName: any;

  constructor(myCustomType: MyCustomType) {
    // implementation...
  }
}
```

Properties and decorators' such as `selector`, `@Input()`, or `@Output()` (same with `inputs` and `outputs`) will probably resonate to you from the time when we overviewed the component decorator spec. Although we haven't mentioned all the possibilities in detail yet, the selector may be declared as one of the following:

- `element-name`: Select by element name
- `.class`: Select by class name
- `[attribute]`: Select by attribute name
- `[attribute=value]`: Select by attribute name and value
- `not(sub_selector)`: Select only if the element does not match the `sub_selector`
- `selector1, selector2`: Select if either `selector1` or `selector2` matches

In addition to this, we will find the `host` parameter, which specifies the events, actions, properties, and attributes pertaining to the host element (that is, the element where our directive takes action) that we want to access from within the directive. We can therefore take advantage of this parameter to bind interaction handlers against the container component or

any other target element of our choice, such as `window`, `document`, or `body`. In this way, we can refer to two very convenient local variables when writing a directive event binding:

- `$event`: This is the current event object that triggered the event.
- `$target`: This is the source of the event. This will be either a DOM element or an Angular directive.

Besides events, we can update specific DOM properties that belong to the host component. We just need to link any specific property wrapped in braces with an expression handled by the directive as a key-value pair in our directive's `host` definition.

Note

The optional `host` parameter can also specify static attributes that should be propagated to a host element, if not present already. This is a convenient way of injecting HTML properties with computed values.

The Angular team has also made available a couple of convenient decorators so that we can more expressively declare our host bindings and listeners straight on the code, like this:

```
@HostBinding('[class.valid]')
isValid: boolean; // The host element will feature class="valid"
                  // is the value of 'isValid' is true.

@HostListener('click', ['$event'])
onClick(e) {
   // This function will be executed when the host // component triggers a
'click' event.
}
```

In the next chapters, we will cover the configuration interface of directives and components in more detail, paying special attention to its life cycle management and how we can easily inject dependencies into our directives. For now, let's just build a simple, yet powerful, directive that will make a huge difference to how our UI is displayed and maintained.

Building a task tooltip custom directive

Let's put in practice some of the settings described above in a custom directive. So far, we have been displaying a tooltip text upon hovering over our pomodoro icons. To do so, we attached a pair of event bindings to the `<pomodoro-task-icons>` element. While this approach is not wrong, the output is a bit verbose and not reusable at all. At some point we may even need to apply the same `[task]` binding elsewhere as well and taking advantage of the same tooltip on mouseover would be quite convenient. Let's automate such functionality in a directive that will get automatically applied to any element featuring a `[task]` attribute, as our `<pomodoro-task-icons>` elements do. This directive will define input properties to refer to that very same property binding and also the target element we will use as a placeholder. If not available, the directive will just do nothing and will not yield any exception whatsoever. When available, the directive will bind `mouseover` and `mouseout` event listeners to the host element (`<pomodoro-task-icons>` in our example). These listeners will toggle the text inside the DOM element represented by the local reference bound to the placeholder property. Before doing so, we will cache the original value in order to reuse it upon moving the mouse out from the element.

The preceding description takes form in the following directive that you should implement before our components in the `pomodoro-tasks.ts` file:

```
@Directive({
  selector: '[task]'
})
class TaskTooltipDirective {
  private defaultTooltipText: string;
  @Input() task: Task;
  @Input() taskTooltip: any;

  @HostListener('mouseover')
  onMouseOver() {
    if(!this.defaultTooltipText && this.taskTooltip) {
      this.defaultTooltipText = this.taskTooltip.innerText;
    }
    this.taskTooltip.innerText = this.task.name;
  }
  @HostListener('mouseout')
  onMouseOut() {
    if(this.taskTooltip) {
      this.taskTooltip.innerText = this.defaultTooltipText;
    }
  }
}
```

Please note the selector in use: `[task]`. We have not configured the more logical `<pomodoro-task-icons>` element or created a new selector of our own. We obviously could have done that, but our goal in this exercise is different. We want to bind a special behavior to any DOM element and component that features a `[task]` attribute with a data binding on it. Precisely because this directive will take action on all elements featuring such property, we can include

it as an input property in the directive implementation itself. Then we just need to provide a way to configure what DOM element will become our tooltip placeholder with the `taskTooltip` input property and we are all set.

As we saw in the previous section, thanks to the `@HostListener()` decorators, we can bind a listener function in our directive to an event occurred in the host component. This time we bound the `mouseover` and `mouseout` event so toggle the text of the target tooltip placeholder, caching its current text beforehand.

In order to see this directive in action, we need to add support for it first at the `PomodoroTasksComponent` decorator:

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [PomodoroIconComponent, TaskTooltipDirective],
  pipes: [FormattedTimePipe, pomodoroQueuedOnlyPipe],
  styleUrls: ['pomodoro-tasks.css'],
  templateUrl: 'pomodoro-tasks.html'
})
class PomodoroTasksComponent {
  // No more changes apply
}
```

Now, we can update our `pomodoro-tasks.html` template:

```
<p>
  <span *ngFor="#queuedTask of tasks | pomodoroQueuedOnly: true">
    <pomodoro-task-icons
      [task]="queuedTask"
      [taskTooltip]="tooltip"
      size="50">
    </pomodoro-task-icons>
  </span>
</p>
<p #tooltip>Mouseover for details</p>
```

One of the most exciting takeaways of this code example is the low code footprint required for extending elements with this new functionality, and its huge reusability.

After all the latest changes, reload your browser, toggle any task, move your mouse over the newly rendered pomodoro icon and... voilà!

A word about naming conventions for custom directives and pipes

Talking about reusability, the common convention is to prepend a custom prefix to the selector. This prevents conflicts with other selectors defined by other libraries we might be using in our project. Same applies to Pipes as well, as we highlighted already when introducing our very first custom pipe.

We have used `pomodoro` as our custom prefix and will keep on using it throughout the book but I advise to use a shorter but recognizable prefix in your custom directives and pipes' selectors.

Ultimately, it is up to you and the name convention you embrace but it is generally a good idea to establish a naming convention that prevents this from happening. A custom prefix is definitely the easier way.

Summary

Now that we have reached this point, it is fair to say that you know almost everything it takes to build Angular 2 components, which are indeed the wheels and the engine of all Angular 2 applications. In the forthcoming chapters, we will see how we can design our application architecture better, and therefore manage dependency injection throughout our components tree, consume data services, leverage the new Angular router to show and hide components when required, and manage user input and authentication.

Nevertheless, this chapter is the backbone of Angular 2 development, and we hope that you enjoyed it as much as we did when writing about template syntax, component APIs based on properties and events, view encapsulation, pipes, and directives. Now, get ready to assume new challenges—we are about to move from learning how to write components to discovering how we can use them to build bigger applications, while enforcing good practices and rational architectures. We will see all this in the next chapter.

Chapter 5. Building an Application with Angular 2 Components

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 component logic in a unique file is definitely not the way to go. Our application might become unmaintainable very soon and, as we will see later in the chapter, we would be missing the advantages that Angular's dependency management mechanism can bring to the game.

In this chapter, we will see how to build application architectures based on trees of components, and how the new Angular 2 dependency injection mechanism will help us to declare and consume our dependencies across the application with minimum effort and optimal results.
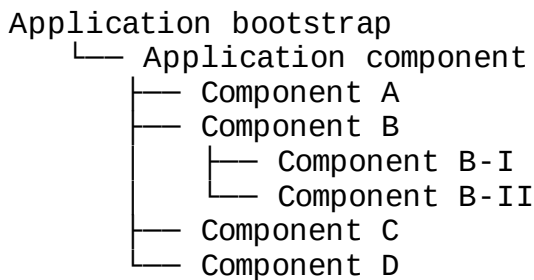
In this chapter, we will cover these topics:

- Best practices for directory structures and naming conventions
- Different approaches to dependency injection
- Injecting dependencies into our custom types
- Overriding global dependencies throughout the component tree
- Interacting with the host component
- Overviewing the directive life cycle

Introducing the component tree

Modern web applications based on web component architectures often conform to a sort of tree hierarchy, wherein the top main component (usually dropped somewhere in the main HTML index file) acts as a global placeholder where child components turn into hosts for other nested child components, and so on and so forth.

There are obvious advantages to this approach. On one hand, reusability does not get compromised and we can reuse components throughout the component tree with little effort. Secondly, the resulting granularity reduces the burden required for envisioning, designing, and maintaining bigger applications. We can simply focus on a single piece of UI and then wrap its functionality around new layers of abstraction until we wrap up a full-blown application from the ground up.

Alternatively, we can approach our web application the other way around, and start from a more generic functionality just to end up breaking down the app into smaller pieces of UI and functionality, which become our web components. The latter has become the most common approach when building component-based architectures. We will stick to it for the rest of the book, undertaking architectures as the one depicted here:

```
Application bootstrap
    └── Application component
        ├── Component A
        ├── Component B
        │   ├── Component B-I
        │   └── Component B-II
        ├── Component C
        └── Component D
```
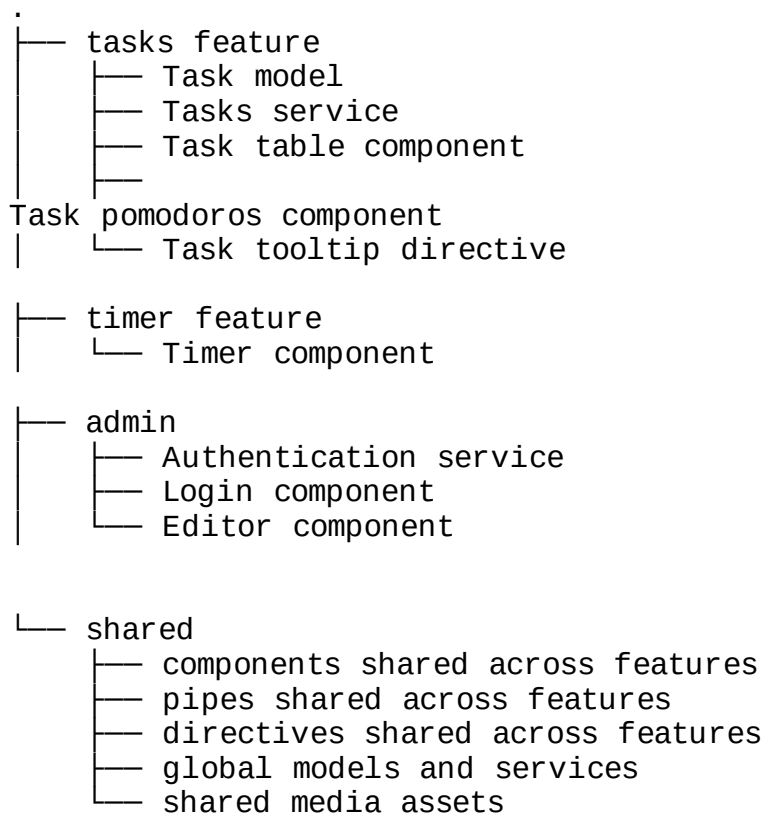
For the sake of clarity, this chapter will just borrow the code we wrote in the previous chapters, and we will deconstruct it into a component hierarchy. We will also allocate some room in the resulting application for all the supporting classes and models required to give shape to our pomodoro tool. This will turn into a perfect opportunity to learn the intricacies of the dependency injection machinery baked into Angular 2, as we will see later in this chapter.

Common conventions for scalable applications

In all fairness, we have already tackled a good number of the common concerns that modern web developers confront when building applications, small and large alike, nowadays. Therefore, it makes sense to define an architecture that will separate the aforementioned concerns into separate domain folders, catering to media assets and shared code units.
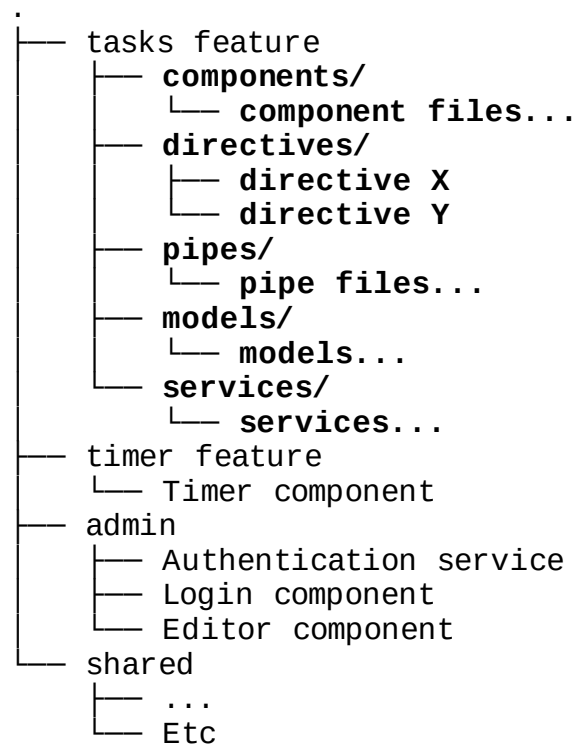
At the time of writing, a commonly agreed pattern for defining project directories embraces the idea of structuring files by features, or contexts. Sometimes, two contexts may require sharing the same entities, and that is fine (as long as it does not become a common thing in our project, which would denote a serious design issue). The following example, applied to our previous work on pomodoro components, depicts this scheme:

```
.
├── tasks feature
│       ├── Task model
│       ├── Tasks service
│       ├── Task table component
│       ├──
Task pomodoros component
│       └── Task tooltip directive

├── timer feature
│       └── Timer component

├── admin
│       ├── Authentication service
│       ├── Login component
│       └── Editor component


└── shared
        ├── components shared across features
        ├── pipes shared across features
        ├── directives shared across features
        ├── global models and services
        └── shared media assets
```
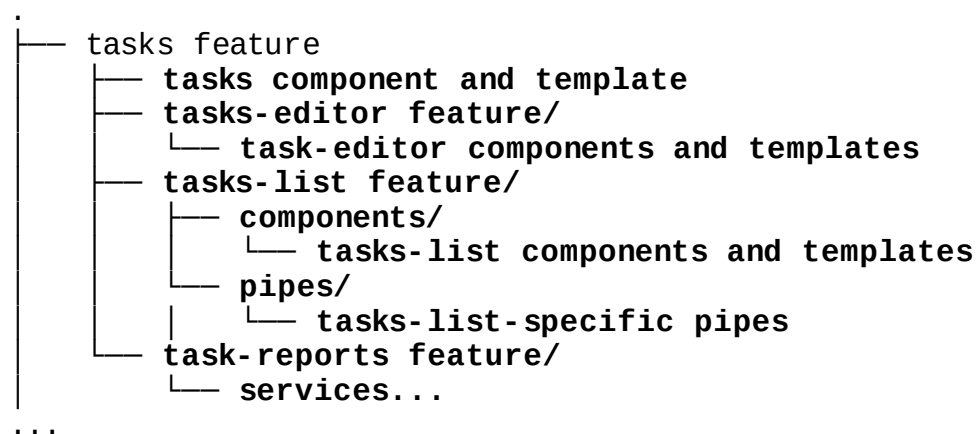
As we can see, the first step is to define the different *features* our application needs, keeping in mind that each one should make sense on its own in isolation from the others. Once we define the set of features required, we will create a folder for each one. Each folder will be filled then with the components, directives, pipes, models, and services that shape the feature it represents. Always remember the principles of encapsulation and reusability when defining your features set.

If the number of files required for any given feature exceeds a logical threshold, then it is fine to organize things a bit and split our files into different folders by *type*. Let's figure out that our *tasks* feature has grown out of control and we have up to 30 files in the folder, between components, directives, pipes, services, test specs, and the like. Identifying code units would

become a burden, so applying an additional layer or organization by type would definitely help:

```
.
├── tasks feature
│   ├── components/
│   │   └── component files...
│   ├── directives/
│   │   ├── directive X
│   │   └── directive Y
│   ├── pipes/
│   │   └── pipe files...
│   ├── models/
│   │   └── models...
│   └── services/
│       └── services...
├── timer feature
│   └── Timer component
├── admin
│   ├── Authentication service
│   ├── Login component
│   └── Editor component
└── shared
    ├── ...
    └── Etc
```

As we can see, it is perfectly fine to have in the same project feature folders with all files at the same level and feature folders containing an additional level of nesting by type. Where to set the threshold is up to you but common sense says that 12 or 15 code units in the same feature folder make a good case for an additional nesting level based on types. We can also combine them by introducing additional nesting levels based on multiple features that fall under the umbrella of an upper context as well, hosting its implementation in a sub-tree by type:

```
.
├── tasks feature
│   ├── tasks component and template
│   ├── tasks-editor feature/
│   │   └── task-editor components and templates
│   ├── tasks-list feature/
│   │   ├── components/
│   │   │   └── tasks-list components and templates
│   │   └── pipes/
│   │       └── tasks-list-specific pipes
│   └── task-reports feature/
│       └── services...
...
```

File and module naming conventions

Each one of our feature folders will host a wide range of files so we need a consistent naming convention to prevent filename collisions while we ensure that the different code units are easy to locate.

The following list summarizes the current conventions enforced by the community:

- Each file should contain a single code unit. Simply put, each component, directive, service, pipe, and so on should live in its own file. This way, we contribute to a better organization of code.
- Files and directories are named in `lower-kebab-case`.
- Files representing components, directives, pipes, and services should append a type suffix to their name: `video-player.ts` will become `video-player.component.ts`.
- Any component's external HTML template or CSS style sheet filename will match the component filename, including the suffix. Our `video-player.component.ts` might be accompanied by `video-player.component.css` and `video-player.component.html`.
- Directive selectors and pipe names are `camelCased`, while component selectors are `lower-kebab-cased`. Plus, it is strongly advised to add a custom prefix of our choice to prevent name collisions with other component libraries. For example, following up our video player component, it may be represented as `<vp-video-player>`, where `vp-` (which stands for *video-player*) is our custom prefix.
- Modules are named by following the rule of taking a `PascalCased` self-descriptive name, plus the type it represents. For example, if we see a module named `VideoPlayerComponent`, we can easily tell it is a component. The custom prefix in use for selectors (`vp-` in our example) should not be part of the module name.

Models and interfaces require special attention though. Depending on your application architecture, model types will feature more or less relevance. Architectures such as MVC, MVVM, Flux, or Redux tackle models from different standpoints and grades of importance. Ultimately, it will be up to you and your architectural design pattern of choice to approach models and their naming convention in one way or another. This book will not be opinionated in that sense, although we do enforce interface models in our example application and will create modules for them.

Ensuring seamless scalability with facades or barrels

Each component and shared context of business logic in our application is intended to integrate with the other pieces of the puzzle in a simple and straightforward way. Clients of each subdomain are not concerned about the internal structure of the subdomain itself. If our *timer* feature, for example, evolves to the point of having two dozen components and directives that need to be reorganized into different folder levels, external consumers of its functionalities should remain unaffected.

This can be done using facade modules that conceal the internal structure of the code by exposing always the same layer of endpoints, hiding the implementation details outside the boundaries of the feature.

In our previous example, the *tasks* feature evolves from being a handful of files inside the same folder into a type-driven set of folders. What if we are using several of its components elsewhere in our application? No worries! A facade module like this would definitely help:

app/tasks/tasks.ts

```
import TaskComponent from './task.component';
import TaskDetailsComponent from './task-details.component';

export {
  TaskComponent,
  TaskDetailsComponent
}
```

Then, we can import any of these components from any other distant corner of our application, like this:

app/example/example.ts

```
import { TaskDetailsComponent } from '../tasks/tasks'
```

What if the *tasks* feature keeps growing and needs to be refactored into different folders? We would just update the path references in our `app/tasks/tasks.ts` facade module, and our client code at `app/example/example.ts` would remain the same.

In the Angular lingo, this kind of design pattern also receives the name of barrel. In Angular's own words:

> A **barrel** is an Angular library module consisting of a logical grouping of single-purpose modules such as `Component` and `Directive`.

Take that into account in order to avoid confusion when bumping into this term in the future. Barrels are also usually grouped and distributed in larger packages named *bundles*. An example of this is the `angular2/bundles/router.js` bundle, for instance. We will not create

packaged bundles in this book, but we will thoroughly use the ones that come with Angular 2 when implementing HTTP connection, routing, or animation functionalities.

How dependency injection works in Angular 2

As our applications grows and evolves, each one of our code entities will internally require instances of other objects, which are better known as *dependencies* in the world of software engineering. The action of passing such dependencies to the dependent client is known as *injection,* and it also entails the participation of another code entity, named the *injector*. The injector will take responsibility for instantiating and bootstrapping the required dependencies so they are ready for use from the very moment they are successfully injected in the client. This is very important since the client knows nothing about how to instantiate its own dependencies and is only aware of the interface they implement in order to use them.

Angular 2 features a top-notch dependency injection mechanism to ease the task of exposing required dependencies to any entity that might exist in an Angular 2 application, regardless of whether it is a component, a directive, a pipe, or any other custom service or provider object. In fact, as we will see later in this chapter, any entity can take advantage of dependency injection (usually referred to as DI) in an Angular 2 application. Before delving deeper into the subject, let's look at the problem that Angular's DI is trying to address.

Let's figure out we have a music player component that relies on a playlist object to broadcast music to its users:

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

@Component({
  selector: 'music-player',
  templateUrl: './music-player.component.html'
})
class MusicPlayerComponent {
  playlist: Playlist;

  constructor() {
    this.playlist = new Playlist();
  }
}
```

The `Playlist` type could be a generic class that returns in its API a random list of songs or whatever. That is not relevant now, since the only thing that matters is that our `MusicPlayerComponent` entity does need it to deliver its functionality. Unfortunately, the implementation above means that both types are tightly coupled, since the component instantiates the playlist *within* its own constructor. This prevents us from altering, overriding, or mocking up in a neat way the `Playlist` class if required. It also entails that a new `Playlist` object is created every time we instantiate a `MusicPlayerComponent`. This might be not desired in certain scenarios, especially if we expect a singleton to be used across the application and thus keep track of the playlist's state.

Dependency injection systems try to solve these issues by proposing several patterns, and the

constructor injection pattern is the one enforced by Angular 2. The previous piece of code could be rethought like this:

```
@Component({
  selector: 'music-player',
  templateUrl: './music-player.component.html'
})
class MusicPlayerComponent {
  playlist: Playlist;

  constructor(playlist: Playlist) {
    this.playlist = playlist;
  }
}
```

Now, the `Playlist` is instantiated outside our component. On the other hand, the `MusicPlayerComponent` expects such an object to be already available before the component is instantiated so it can be injected through its constructor. This approach gives us the opportunity to override it or mock it up if we wish.

Basically, this is how dependency injection, and more specifically the constructor injection pattern, works. However, what has this to do with Angular 2? Does Angular's dependency injection machinery work by instantiating types by hand and injecting them through the constructor? Obviously not, mostly because we do not instantiate components by hand either (except when writing unit tests). Angular features its own dependency injection framework, which can be used as a standalone framework by other applications, by the way.

The framework offers an actual injector that can introspect the tokens used to annotate the parameters in the constructor and return a singleton instance of the type represented by each dependency, so we can use it straight away in the implementation of our class, as in the previous example. The injector ignores how to create an instance of each dependency, so it relies on the list of providers registered upon bootstrapping the application. Each one of those providers actually *provides* mappings over the types marked as application dependencies. Whenever an entity (let's say a component, a directive, or a service) defines a token in its constructor, the injector searches for a type matching that token in the pool of registered providers for that component. If no match is found, it will then delegate the search on the parent component's provider, and will keep conducting the provider's lookup upwards until a provider resolves with a matching type or the top component is reached. Should the provider lookup finish with no match, Angular 2 will throw an exception.

Note

The latter is not exactly true, since we can mark dependencies in the constructor with the `@Optional` parameter decorator, in which case Angular 2 will not throw any exception and the dependency parameter will be injected as `null` if no provider is found. The topmost component is not the last dead-end of the provider lookup, since we can also declare global dependencies in the `bootstrap()` function, as we will see later in this chapter.

Whenever a provider resolves with a type matching that token, it will return such type as a singleton, which will be therefore injected by the injector as a dependency. In fairness, the provider is not just a collection of key/value pairs coupling tokens with previously registered types, but a factory that instantiates these types and also instantiates each dependency's very own dependencies as well, in a sort of recursive dependency instantiation.

So, instead of instantiating the `Playlist` object manually, we could do this:

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

@Component({
  selector: 'music-player',
  templateUrl: './music-player.component.html',
  providers: [Playlist]
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```

The `providers` property of the `@Component` decorator is the place where we can register dependencies on a component level. From that moment onwards, these types will be immediately available for injection at the constructor of that component and, as we will see next, at its own child components as well.

Injecting dependencies across the component tree

We have seen that the provider lookup is performed upwards until a match is found. A more visual example might help, so let's figure out that we have a music app component that hosts in its `directives` property (and hence its template) a music library component with a collection of all our downloaded tunes which also hosts, in its own `directives` property and template, a music player component so we can playback any of the tunes in our library.

```
.
├── MusicAppComponent()
│   └── MusicLibraryComponent()
│       └── MusicPlayerComponent()
...
```

Our music player component requires an instance of the `Playlist` object we mentioned before, so we declare it as a constructor parameter, conveniently annotated with the `Playlist` token.

```
.
├── MusicAppComponent()
│   └── MusicLibraryComponent()
│       └── MusicPlayerComponent(playlist: Playlist)
...
```

When the `MusicPlayerComponent` entity is instantiated, the Angular DI mechanism will go through the parameters in the component constructor with special attention to their type annotations. Then, it will check if that type has been registered in the component's `provider` property of the component decorator configuration. The code is as follows:

```
@Component({
  selector: 'music-player',
  providers: [Playlist]
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```

But, what if we want to reuse the `Playlist` type in other components throughout the same component tree? Maybe the `Playlist` type contains functionalities in its API that are required by different components at once across the application. Do we have to declare the token in the provider's property for each one? Fortunately not, since Angular 2 anticipates that necessity and brings transversal dependency injection through the component tree.

Note

In the previous section, we mentioned that components conduct a provider lookup upwards. This is because each component has its own built-in injector, which is specific to it. Nevertheless, that injector is in reality a child instance of the parent's component injector (and so on so forth), so it is fair to say that an Angular 2 application has not a single injector, but

many instances of the same injector, so to say.

We need to extend the injection of the `Playlist` object to other components in the component tree in a quick and reusable fashion. Knowing beforehand that components perform a provider lookup starting from itself and then passing up the request to its parent component's injectors, we can then address the issue by registering the provider in the parent component, or even the top parent component, so the dependency will be available for injection for each and every child component found underneath it. In this sense, we could register the `Playlist` object straight at `MusicAppComponent`, regardless it might not need it for its own implementation:

```
@Component({
  selector: 'music-app',
  providers: [Playlist],
  directives: [MusicLibraryComponent],
  template: '<music-library></music-library>'
})
class MusicAppComponent {}
```

The immediate child component might not require the dependency for its own implementation either. Since it has been already registered in its parent `MusicAppComponent` component, there is no need to register it there again.

```
@Component({
  selector: 'music-library',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>'
})
class MusicLibraryComponent {}
```

We finally reach our music player component, but now it no longer features the `Playlist` type as a registered token in its `providers` property. In fact, our component does not feature a `providers` property at all. It no longer requires this, since the type has been already registered somewhere above the component's hierarchy, being immediately available for all child components, no matter where they are.

```
@Component({
  selector: 'music-player'
})
class MusicPlayerComponent {
  constructor(public playlist: Playlist) {}
}
```
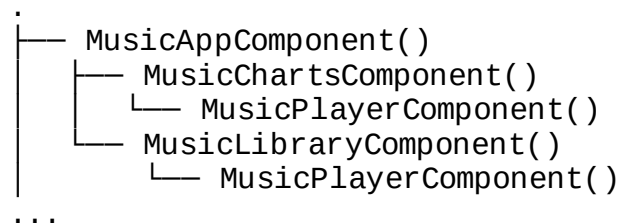
Now, we see how dependencies are injected down the component hierarchy and how the provider lookup is performed by components just by checking their own registered providers and bubbling up the request upwards in the component tree. However, what if we want to constrain such injection or lookup actions?

Restricting dependency injection down the component tree

In our previous example, we saw how the music app component registered the `Playlist` token in its `providers` collection, making it immediately available for all child components. Sometimes, we might need to constrain the injection of dependencies to reach only those directives (and components) that are immediately next to a specific component in the hierarchy. We can do that by registering the type token in the `viewProviders` property of the component decorator, instead of using the `providers` property we've seen already. In our previous example, we can restrain the downwards injection of `Playlist` one level only:

```
@Component({
  selector: 'music-app',
  viewProviders: [Playlist],
  directives: [MusicLibraryComponent],
  template: '<music-library></music-library>'
})
class MusicAppComponent {}
```

We are informing Angular 2 that the `Playlist` provider should only be accessible by the injectors of the directives and components located in the `MusicAppComponent` view, but not for the children of such components. The use of this technique is exclusive of components, since only they feature views.

Restricting provider lookup

Just like we can restrict dependency injection, we can constrain dependency lookup to the immediate upper level only. To do so, we just need to apply the `@Host()` decorator to those dependency parameters whose provider lookup we want to restrict:

```
Import { Component, Host } from '@angular/core';

@Component({
  selector: 'music-player'
})
class MusicPlayerComponent {
  constructor(@Host() playlist: Playlist) {}
}
```

According to the preceding example, the `MusicPlayerComponent` injector will look up for a `Playlist` type at its parent component's providers collection (`MusicLibraryComponent` in our example) and will stop there, throwing an exception because `Playlist` has not been returned by the parent's injector (unless we also decorate it with the `@Optional()` parameter decorator).

Overriding providers in the injector hierarchy

We've seen so far how Angular's DI framework uses the dependency token to introspect the type required and return it right from any of the provider sets available along the component hierarchy. However, we might need to override the class instance corresponding to that token in certain cases where a more specialized type is required to do the job. Angular provides special tools to override the providers or even implement factories that will return a class instance for a given token, not necessarily matching the original type.

We will not cover all the use cases in detail here, but let's look at a simple example. In our example, we assumed that the `Playlist` object was meant to be available across the component tree for use in different entities of the application. What if our `MusicAppComponent` directive hosts another component whose child directives require a more specialized version of the `Playlist` object? Let's rethink our example:

```
.
├── MusicAppComponent()
│   ├── MusicChartsComponent()
│   │   └── MusicPlayerComponent()
│   └── MusicLibraryComponent()
│       └── MusicPlayerComponent()
...
```

This is a bit contrived example but will definitely help us to understand the point of overriding dependencies. The `Playlist` instance object is available right from the top component downwards. The `MusicChartsComponent` directive is a specialized component that caters only for music featured in the top seller's charts and hence its player must playback big hits only, regardless of the fact it uses the same component as `MusicLibraryComponent`. We need to ensure that each player component gets the proper playlist object, and this can be done at the `MusicChartsComponent` level by overriding the object instance corresponding to the `Playlist` token. The following example depicts this scenario, leveraging the use of the `provide` function:

```
import { Component, provide } from '@angular/core';
import { Playlist } from './playlist';
import { TopHitsPlaylist } from './top-hits-playlist';

@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [provide(Playlist, { useClass: TopHitsPlaylist })]
})
class MusicChartsComponent {}
```

The `provide()` function creates a provider mapped to the token specified in the first argument (`Playlist` in this example) and the implementation configured in the second argument, which in this case points to using the `TopHitsPlaylist` type as the reference class.

We could refactor the block of code to use `viewProviders` instead, so we ensure that (if required) the child entities still receive an instance of `Playlist` instead of `TopHitsPlaylist`. Alternatively, we can go the extra mile and use a factory, to return the specific object instance we need depending on other requirements. The following example will return a different object instance for the `Playlist` token depending on the `evaluation` of a Boolean `condition` variable:

```
@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [
    provide(Playlist, { useFactory: () => {
        if(condition) {
          return new TopHitsPlaylist();
        } else {
          return new Playlist();
        }
      }
    })
  ]
})
class MusicChartsComponent {}
```

Moving out from the preceding pseudo-code example, how can we provide a better logic flow when using the `useFactory` function? It turns out that the method signature can take arguments that operate pretty much the same as dependencies do when in the constructor of any given Angular entity. We just need to point Angular to the type each argument token of the `useFactory` lambda function has by declaring them in the `deps` property as follows:

```
@Component({
  selector: 'music-charts',
  directives: [MusicPlayerComponent],
  template: '<music-player></music-player>',
  providers: [
    ConditionalService,
    provide(Playlist, { useFactory: (conditionalService) => {
        if(conditionalService.isTopHits) {
          return new TopHitsPlaylist();
        } else {
          return new Playlist();
        }
      },
      deps: [ConditionalService]
    })
  ]
})
class MusicChartsComponent {}
```

In the preceding example, we are injecting an object instance of an imaginary `ConditionalService` class, which exposes a Boolean property named `isTopHits` that will inform about the playlist to be used. Keep in mind that these types will have to be registered as

well, either in the `providers` property of the current component or at any of its parent
components.

Extending injector support to custom entities

Directives and components require dependencies to be introspected, resolved, and injected. Other entities such as service classes often require quite such functionality too. In our example, our `Playlist` class might rely on a dependency on a HTTP client to communicate with a third party to fetch the songs. The action of injecting such dependency should be as easy as declaring the annotated dependencies in the class constructor and have an injector ready to fetch the object instance by inspecting the class provider or any other provider available somewhere.

It is only when we think hard about the latter that we realize there is a gap in this idea: custom classes and services do not belong to the component tree. Hence, they do not benefit from anything such as a built-in injector or a parent injector. We cannot even declare a `providers` property, since we do not decorate these types of class with a `@Component` or `@Directive` decorator. Let's take a look at an example:

```
class Playlist {
  songs: string[];
  constructor(songsService: SongsService) {
    this.songs = songsService.fetch();
  }
}
```

We might try the above in the hope of having Angular 2's DI mechanism introspecting the `songsService` parameter of the `Playlist` class constructor when instantiating this class in order to inject it into `MusicPlayerComponent`. Unfortunately, the only thing we will eventually get is an exception like this:

```
Cannot resolve all parameters for Playlist(?). Make sure they all have valid
type or annotations.
```

This is kind of misleading, since all constructor parameters in `Playlist` have been properly annotated, right? As we said before, the Angular DI machinery resolves dependencies by introspecting the types of the constructor parameters. To do so, it needs some metadata to be created beforehand. Each and every Angular entity class decorated with a decorator features this metadata as a by-product of the way TypeScript compiles the decorator configuration details. However, dependencies that also require other dependencies have no decorator whatsoever and no metadata is then created for them. This can be easily fixed thanks to the `@Injectable()` decorator, which will give visibility to these service classes for the DI mechanism:

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

@Injectable()
class Playlist {
  songs: string[];
  constructor(songsService: SongsService) {
    this.songs = songsService.fetch();
  }
```

```
}
```

You will get used to introducing that decorator in your service classes, since they will quite often rely on other dependencies not related to the component tree in order to deliver the functionality.

Tip

It is actually a good practice to decorate all your service classes with the `@Injectable()` decorator, irrespective of whether its constructor functions have dependencies or not. This way, we prevent errors and exceptions because of skipping this requirement once the service class grows and requires more dependencies in the future.

Initializing applications with bootstrap()

As we have seen in this chapter, the dependency lookup bubbles up until the first component at the top. This is not exactly true, since there is an additional step that the DI mechanism will check on: the `bootstrap()` function.

As far as we know, we use the `bootstrap()`function to kickstart our application by declaring in its first argument the root component that initiates the application's component tree. However, the bootstrap function takes a second argument in the form of a providers array, where we can explicit dependencies as well, that will become available throughout the injector tree. However, this is a bad practice because it couples the availability of any provider to the application itself, constraining the encapsulation and reusability of our components, moreover when the `bootstrap` initialization function is platform-specific.

Where shall we declare our application dependencies then? Always use our top root component instead. The providers argument of the `bootstrap` function should only be used when we need to override existing Angular 2 providers on an application level, leveraging the `provide()` function, for instance.

Always keep in mind that we can have multiple root components, each one of them the result of multiple executions of the `bootstrap()` function declaring a different root component each time. Each one of these root components will feature its own set of injectors and service singletons, with no relationship whatsoever amongst them.

Switching between development and production modes

Angular 2 applications are bootstrapped and initialized by default in *development mode*. In the development mode, the Angular 2 runtime will throw warning messages and assertions to the browser console. While this is quite useful for debugging our application, we do not want those messages to be displayed when the application is in production. The good news is that the development mode can be disabled in favor of the more silent production mode. This action is usually performed before bootstrapping our application:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableProdMode } from 'angular/core';
import AppComponent from './app.component';

enableProdMode();

bootstrap(AppComponent, []);
```

Enabling Angular 2's built-in change detection profiler

We can also access advanced tools from the browser console by enabling the Angular Debug Tools. To do so, just import the `enableDebugTools` function, which is specific in to the browser platform, and execute it as soon as you get hold of an instance of the component you want to profile. The code is as follows:

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import { enableDebugTools } from '@angular/platform-browser';
import { ComponentRef } from 'angular/core';
import AppComponent from './app.component';

// The bootstrap() function returns a promise with
// a reference to the bootstrapped component
bootstrap(AppComponent, []).then((ref: ComponentRef) => {
  enableDebugTools(ref);
});
```

When the `enableDebugTools()` function is triggered, you just need to follow the following steps to access the change detection profiler:

1. Open the browser dev tools and switch to the console view.
2. Type `ng.profiler.timeChangeDetection({record: true})` and then press *Enter*.
3. The Angular 2 runtime will exercise change detection in a loop and will print the average amount of time a single round of change detection takes for the current state of the UI. A CPU profile recording will be conducted while the change detector is exercised.

Hopefully, the debug tools will be fleshed out with more functionalities in the future. Stay tuned and refer to the official documentation.

Introducing the Pomodoro App directory structure

In the previous chapters and sections in this chapter, we have seen different approaches and good practices for laying out Angular 2 applications. These guidelines encompassed from naming conventions to pointers about how to organize files and folders. From this point onwards, we are going to put all this knowledge to practice by refactoring all the different interfaces, components, directives, pipes, and services in an actual Angular 2 architecture, conforming to the most commonly agreed community conventions.

By the end of this chapter, we will have a final application layout that wraps everything we have seen so far in the following site architecture:

```
.
├── app/
│   ├── shared/
│   │   ├── assets/ ← Global CSS or image files are stored here
│   │   ├── directives/
│   │   ├── interfaces/
│   │   ├── pipes/
│   │   ├── services/
│   │   └── shared.ts ← facade for the 'shared' context
│   ├── tasks/
│   │   ├── (tasks-related components and directives)
│   │   └── tasks.ts ← facade for the 'tasks' context
│   ├── timer/
│   │   ├── (timer-related components and directives)
│   │   └── timer.ts ← facade for the 'timer' context
│   │
│   ├── app.component.ts ← top root application component
│   └── main.ts ← here we bootstrap the top root component
│
├── index.html
├── package.json
├── tsconfig.json
└── typings.json
```

It is easy to understand the whole rationale of the project. Now, we will put together an application that features two main contexts: a *timer* feature and a *tasks listing* feature. Each feature can encompass a different range of components, pipes, directives, or services. The inner implementation of each feature is opaque to the other features or contexts. Each feature context exposes a facade that exports the pieces of functionality (that is, the component, one or many) that each context delivers to the upper-level context or application. All the other pieces of functionality (inner services or directives) are concealed from the rest of the application.

It is fair to say that it is difficult to draw a line in the sand differentiating what belongs to a specific context or another. Sometimes, we build pieces of functionality, such as certain directives or pipes, which can be reused throughout the application. So, locking them down to

a specific context does not make much sense. For those cases, we do have the *shared* context, where we store any code unit which is meant to be reusable at an application level, apart from media files such as style sheets or bitmap images that are component-agnostic.

The main `app.component.ts` file contains and exports the application root component, which declares and registers in its own injector the dependencies required by its child components. As you know already, all Angular 2 applications must have at least one root component, initialized by the `bootstrap()` function. This operation is actually performed in the `main.ts` file, which is fired by the `index.html` file.

Defining a component or a group of related components within a context like this improves reusability and encapsulation. The only component that is tightly coupled with the application is the top root component, whose functionality is usually pretty limited and entails basically rendering the other child components in its template view or acting as a router component, as we will see in the next chapters.

The last bit of the puzzle is the JSON files that contain the TypeScript compiler, typings, and `npm` configuration. Since versioning on the Angular 2 framework keeps evolving, we will not look at the actual content of these files here. You are supposed to know their purpose, but some specifics such as the peer dependency versions change quite often so you better refer to the book's GitHub repository for the latest up-to-date version of each one. The `package.json` file requires a special mention though. There are a few common industry conventions and popular seed projects, like the one provided by the Angular official site itself. We have provided several `npm` commands to ease the overall installation process and the development endeavor:

1. Go to https://github.com/deeleman/angular2-essentials and download `package.json`, `typings.json` and `tsconfig.json` into a folder of your choice. We recommend you to download `index.html` as well.
2. Open up your console in the folder where you saved the preceding files and run `npm install`. Angular 2 and all its peer dependencies and required typings will be downloaded and installed in the project workspace.
3. Now, you can just run `npm start` in the console, enable the TypeScript compiler in watch mode, and fire a local server pointing to this project folder. We recommend you to create the application files before executing this command or an exception will be triggered.

Refactoring our application the Angular 2 way

In this section, we will split the code we created in Chapters 1, 3, and 4 into code units, following the **single responsibility** principle. So, do not expect many changes in the code, apart from allocating each module in its own dedicated file. This is why we will focus more on how to split things rather than explaining each module, whose purpose you should know already. In any event, we will take a minute to discuss changes if required.

Let's begin by creating in your work folder the same directory structure we saw in the previous section. We will populate each folder with files on the go.

The shared context

The shared context is where we store any module whose functionality is meant to be used by not one but many contexts at once, as it is agnostic to those contexts as well. A good example is the pomodoro bitmap we've been using to decorate our components, which should be stored in the `app/shared/assets/img` path (please do save it there, by the way).

Another good example is the interfaces that model data, mostly when their schema can be reused across a different context of functionality. For instance, when we defined the `QueuedOnlyPipe` in [Chapter 3](#), *Implementing Properties and Events in Our Components,* we actioned only over the `queued` property of items in the recordset. We can then seriously consider implementing a `Queued` interface that we can use later on to provide type-checking for modules that feature that property. This will make our pipes more reusable and model-agnostic. The code is as follows:

app/shared/interfaces/queuable.ts

```
interface Queueable {
  queued: boolean;
}

export default Queueable;
```

Tip

Pay attention to this workflow: first we define the module corresponding to this code unit, and then we export it, flagging it as default so we can import it by name from elsewhere. Interfaces need to be exported this way, but for the rest of the book we will usually declare the module and export it in the same statement.

With this interface in place, we can now safely refactor the `QueuedOnlyPipe` to make it fully agnostic from the `Task` interface so that it is fully reusable on any context where a recordset, featuring items implementing the `Queued` interface, needs to be filtered, regardless of what they represent. The code is as follows:

app/shared/pipes/queued-only.pipe.ts

```
import { Pipe, PipeTransform } from 'angular/core';
import { Queueable } from '../shared';

@Pipe({
  name: 'pomodoroQueuedOnly',
  pure: false
})
export default class QueuedOnlyPipe implements PipeTransform {
  transform(
    queueableItems: Queueable[],
    ...args): Queueable[] {
    return queueableItems.filter((queueableItem: Queueable) => {
```

```
      return queueableItem.queued === args[0]
    });
  }
}
```

As you can see, each code unit contains a single module. This code unit conforms to the naming conventions set for Angular 2 filenames, clearly stating the module name in camel case, plus the type suffix (`.pipe` in this case). The implementation does not change either, apart from the fact that we have annotated all queue-able items with the `Queuable` type, instead of the `Task` annotation we had earlier. Now, our pipe can be reused wherever a model implementing the `Queued` interface is present.

However, there is something that should draw your attention: we're not importing the `Queuable` interface from its source location, but from a file named `shared.ts` located in the upper level. This is the facade file for the *shared* context, and we will expose all public shared modules from that file not only to the clients consuming the *shared* context modules, but to those inside the *shared* context as well. There is a case for this: if any module within the *shared* context changes its location, we need to update the facade so that any other element referring to that module within the same context remains unaffected, since it consumes it through the facade. This is actually a good moment to start beefing up our very first facade then:

app/shared/shared.ts

```
import Queueable from './interfaces/queueable';

export {
  Queueable
};
```

As you can see, facades have no business logic implementation and, in their simplest incarnation, are just a summarized block of imports publicly exposed in a single export. Now that we have a working `Queuable` interface and a facade, we can create the other interface we will require throughout the book, corresponding to the `Task` entity, along with the other pipe we required—both exposed through the facade as well:

app/shared/interfaces/task.ts

```
import { Queueable } from '../shared';

interface Task extends Queueable {
  name: string;
  deadline: Date;
  pomodorosRequired: number;
}

export default Task;
```

We implement an interface onto another interface in TypeScript by using `extends` (instead of

implements). Now, for the `FormattedTimePipe`:

app/shared/pipes/formatted-time.pipe.ts

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

@Pipe({
  name: 'pomodoroFormattedTime'
})
export default class FormattedTimePipe implements PipeTransform {
  transform(totalMinutes: number): string {
    let minutes: number = totalMinutes % 60;
    let hours: number = Math.floor(totalMinutes / 60);
    return `${hours}h:${minutes}m`;
  }
}
```

Obviously, both modules will be made available publicly from the facade, as we did previously.

Services in the shared context

There is no rule of thumb for services with regard to where they should go. Some schools of thought assume that services are mere data and logic providers and, as such, should be agnostic of what actually consumes them, irrespective of whether it is a component, directive, or any other service. Services become then first-class citizens that can easily be promoted to their own project workspace for greater reusability across different projects. Some other practitioners prefer to bind services to the feature context they belong to, if only a single context applies, favoring encapsulation. Ultimately, it will depend on the level of reusability versus encapsulation you aim to achieve in your application, and what entity actually makes use of the data and logic of those services.

We built a data service in the previous chapter to serve a tasks dataset to populate our data table with. As we will see later in this book, the data service will be consumed by other contexts of the application. So, we will allocate it in the shared context, exposing it through the facade as usual:

app/shared/services/task.service.ts

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

@Injectable()
export default class TaskService {
  public taskStore: Task[] = [];

  constructor() {
    const tasks = [
      {
        name: "Code an HTML Table",
        deadline: "Jun 23 2015",
```

```
        pomodorosRequired: 1
      }, {
        name: "Sketch a wireframe for the new homepage",
        deadline: "Jun 24 2016",
        pomodorosRequired: 2
      }, {
        name: "Style table with Bootstrap styles",
        deadline: "Jun 25 2016",
        pomodorosRequired: 1
      }, {
        name: "Reinforce SEO with custom sitemap.xml",
        deadline: "Jun 26 2016",
        pomodorosRequired: 3
      }
    ];

    this.taskStore = tasks.map(task => {
      return {
        name: task.name,
        deadline: new Date(task.deadline),
        queued: false,
        pomodorosRequired: task.pomodorosRequired
      };
    });
  }
}
```

Please pay attention to how we imported the `Injectable()` decorator and implemented it on our service. It does not require any dependency in its constructor, so other modules depending on this service will not have any issues anyway when declaring it in its constructors. The reason is simple: it is actually a good practice to apply the `@Injectable()` decorator in our services by default to ensure they keep being injected seamlessly as long as they begin depending on other providers, just in case we forget to decorate them then.

Configuring application settings from a central service

In the previous chapters, we hardcoded a lot of stuff in our components: labels, pomodoro durations, plural mappings, and so on. Sometimes, our contexts are meant to have a high level of specificity and and it's fine to have that information there. At other times, we might require more flexibility and a more convenient way to update these settings application-wide. For this example, we will make all the l18n pipes mappings and pomodoro settings available from a central service located in the shared context and exposed, as usual, from the `shared.ts` facade.

app/shared/services/settings.service.ts

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

@Injectable()
export default class SettingsService {
  timerMinutes: number;
  labelsMap: any;
  pluralsMap: any;
```

```
  constructor() {
    this.timerMinutes = 25;
    this.labelsMap = {
      'timer': {
        'start': 'Start Timer',
        'pause': 'Pause Timer',
        'resume': 'Resume Countdown',
        'other': 'Unknown'
      }
    };
    this.pluralsMap = {
      'tasks': {
        '=0': 'No pomodoros',
        '=1': 'One pomodoro',
        'other': '# pomodoros'
      }
    }
  }
}
```

Please note how we expose context-agnostic mapping properties, which are actually namespaced, to better group the different mappings by context.

It would be perfectly fine to split this service into two specific services, one per context, and locate them inside their respective context folders, at least with regard to the l18n mappings. Keep in mind that data such as the time duration per pomodoro will be used across different contexts though, as we will see later in this chapter.

Creating a facade module including a custom providers barrel

With all the latest changes, our `shared.ts` facade should look like this:

app/shared/shared.ts

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,

  SettingsService,
  TaskService
};
```

Our facade exposes interface typings, pipes, and service providers. As we will see when injecting our dependencies globally from the root component, it is actually quite common and convenient to group services (and directives as well, including components in the same group) into grouping alias tokens, usually named after the context name followed by the `_PROVIDERS` suffix, all in uppercase. With regard to the facade, we could introduce this block of code right above the export statement:

```
// import statements remain unchanged above
const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  SettingsService,
  TaskService
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  SettingsService,
```

```
    TaskService,
    SHARED_PROVIDERS
};
```

This way, we can register all our providers through a single token where required. The same applies to directives and components, where the rule of thumb is to export in the context facade a token with the name `{CONTEXTNAME}_DIRECTIVES`. This gives us the opportunity to inject support for all required components and directives from a context in another component by means of a single token. If such context winds up exposing more components and directives in the future or the names of its existing logical modules change, we will not need then to follow the trail of module tokens registered in the `directives` property of our components throughout the application. We will see all this in action further up in this chapter.

Creating our components

With our shared context sorted out, time has come to cater with our other two contexts: timer and tasks. Their names are self-descriptive enough of the scope of their functionalities. Each context folder will allocate the component, HTML view template, CSS, and directive files required to deliver their functionality, plus a facade file that exports the public components of this feature.

The timer context

Our first context is the one belonging to the **timer** functionality, which happens to be the simpler one as well. It comprises a unique component with the countdown timer we built in the previous chapters:

app/timer/timer-widget.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { SettingsService } from '../shared/shared';

@Component({
  selector: 'pomodoro-timer-widget',
  template: `
    <div class="text-center">
      <img src="/app/shared/assets/img/pomodoro.png">
      <h1> {{ minutes }}:{{ seconds  | number: '2.0' }} </h1>
      <p>
        <button (click)="togglePause()" class="btn btn-danger">
        {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
        </button>
      </p>
    </div>`
})
export default class TimerWidgetComponent {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabelKey: string;
  buttonLabelsMap: any;

  constructor(private settingsService: SettingsService) {
    this.buttonLabelsMap = settingsService.labelsMap.timer;
  }

  ngOnInit(): void {
    this.resetPomodoro();
    setInterval(() => this.tick(), 1000);
  }

  resetPomodoro(): void {
    this.isPaused = true;
    this.minutes = this.settingsService.timerMinutes - 1;
    this.seconds = 59;
    this.buttonLabelKey = 'start';
```

```
    }

  private tick(): void {
    if (!this.isPaused) {
      this.buttonLabelKey = 'pause';

      if (--this.seconds < 0) {
        this.seconds = 59;
        if (--this.minutes < 0) {
          this.resetPomodoro();
        }
      }
    }
  }

  togglePause(): void {
    this.isPaused = !this.isPaused;
    if (this.minutes < this.settingsService.timerMinutes || this.seconds < 59) {
      this.buttonLabelKey = this.isPaused ? 'resume' : 'pause';
    }
  }
}
```

As you can see, the implementation is pretty much the same we saw already back in [Chapter 1](#), *Creating Our Very First Component in Angular 2*, with the exception of initializing the component at the init life cycle stage through the `OnInit` interface hook. We leverage the `I18nSelect` pipe to better handle the different labels required for each state of the timer, consuming the label information from the `SettingsService` provider, which is injected by the component injector through an annotated argument in the constructor. Later on in this chapter, we will see where to register that provider. The pomodoro duration in minutes is also consumed from the service, once the latter is bound to a class field.

The component is exported publicly through a facade, saving its client components from having to know the actual path and filename of the component. The code is as follows:

app/timer/timer.ts

```
import TimerWidgetComponent from './timer-widget.component';

const TIMER_DIRECTIVES: any[] = [
  TimerWidgetComponent
];

export {
  TIMER_DIRECTIVES,
  TimerWidgetComponent
};
```

Please note that the `TIMER_DIRECTIVES` alias token is included for our convenience should the component range grow in the future and we wish to take advantage of a single entry point to all components and directives publicly available.

The tasks context

The **tasks** context encompasses some more logic, since it entails two components and a directive. Let's begin by creating the core unit required by `TaskTooltipDirective`:

app/tasks/task-tooltip.directive.ts

```
import { Task } from '../shared/shared';
import { Input, Directive, HostListener } from '@angular/core';

@Directive({
  selector: '[task]'
})
export default class TaskTooltipDirective {
  private defaultTooltipText: string;
  @Input() task: Task;
  @Input() taskTooltip: any;

  @HostListener('mouseover')
  onMouseOver() {
    if(!this.defaultTooltipText && this.taskTooltip) {
      this.defaultTooltipText = this.taskTooltip.innerText;
    }
    this.taskTooltip.innerText = this.task.name;
  }
  @HostListener('mouseout')
  onMouseOut() {
    if(this.taskTooltip) {
      this.taskTooltip.innerText = this.defaultTooltipText;
    }
  }
}
```

The directive keeps all the original functionality in place and just imports the Angular 2 core types and task-typing it requires. Let's look at the `TaskIconsComponent` now:

app/tasks/task-icons.component.ts

```
import { Component, Input, OnInit } from '@angular/core';
import { Task } from '../shared/shared';

@Component({
  selector: 'pomodoro-task-icons',
  template: `<img *ngFor="let icon of icons"
                  src="/app/shared/assets/img/pomodoro.png"
                  width="{{size}}">`
})
export default class TaskIconsComponent implements OnInit {
  @Input() task: Task;
  @Input() size: number;
  icons: Object[] = [];

  ngOnInit() {
    this.icons.length = this.task.pomodorosRequired;
```

```
        this.icons.fill({ name: this.task.name });
    }
}
```

So far so good. Now, let's jump to `TasksComponent`. This component will require some more overhead, since it features external HTML templates and style sheets, which are basically the same we already built back in [Chapter 4](), *Enhancing our Components with Pipes and Directives*:

app/tasks/tasks.component.css

```
h3, p {
    text-align: center;
}
.table {
    margin: auto;
    max-width: 860px;
}
```

app/tasks/tasks.component.html

```
<div class="container text-center">
  <h3>
    {{ queuedPomodoros | i18nPlural: queueHeaderMapping }} for today
    <span class="small" *ngIf="queuedPomodoros > 0">(Estimated time:
    {{ queuedPomodoros * timerMinutes | pomodoroFormattedTime }})
    </span>
  </h3>
  <p>
    <span*ngFor="let queuedTask of tasks | pomodoroQueuedOnly: true">
      <pomodoro-task-icons
        [task]="queuedTask"
        [taskTooltip]="tooltip"
        size="50">
      </pomodoro-task-icons>
    </span>
  </p>
  <p #tooltip [hidden]="queuedPomodoros === 0">Mouseover for details</p>

  <h4>Tasks backlog</h4>
  <table class="table">
    <thead>
      <tr>
        <th>Task ID</th>
        <th>Task name</th>
        <th>Deliver by</th>
        <th>Pomodoros</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let task of tasks; let i = index">
        <th scope="row">{{i}}
          <span *ngIf="task.queued" class="label label-info">
            Queued
```

```
        </span>
      </th>
      <td>{{task.name | slice: 0:35 }}
        <span [hidden]="task.name.length < 35">...</span>
      </td>
      <td>{{task.deadline | date: 'fullDate' }}
        <span *ngIf="task.deadline < today"class="label label-danger">
          Due
        </span>
      </td>
      <td class="text-center">{{task.pomodorosRequired}}</td>
      <td>
        <button type="button" class="btn btn-default btn-xs"
          [ngSwitch]="task.queued"
          (click)="toggleTask(task)">
          <template [ngSwitchWhen]="false">
            <i class="glyphicon glyphicon-plus-sign"></i>
            Add
          </template>
          <template [ngSwitchWhen]="true">
            <i class="glyphicon glyphicon-minus-sign"></i>
            Remove
          </template>
          <template ngSwitchDefault>
            <i class="glyphicon glyphicon-plus-sign"></i>
            Add
          </template>
        </button>
      </td>
    </tr>
  </tbody>
</table>
</div>
```

Please take a moment to check out the naming convention applied to the external component files, whose filename matches the component's own to identify which file belongs to what in flat structures inside a context folder. Also, please note how we removed the main pomodoro bitmap from the template and replaced the hardcoded pomodoro time durations with a variable named `timerMinutes` in the binding expression that computes the time estimation to accomplish all queued tasks. We will see how that variable is populated in the following component class:

app/tasks/tasks.component.ts

```
import { Component, OnInit } from '@angular/core';
import TaskIconsComponent from './task-icons.component';
import TaskTooltipDirective from './task-tooltip.directive';
import {
  TaskService,
  SettingsService,
  Task,
  SHARED_PIPES
} from '../shared/shared';
```

```
@Component({
  selector: 'pomodoro-tasks',
  directives: [TaskIconsComponent, TaskTooltipDirective],
  pipes: [SHARED_PIPES],
  styleUrls: ['app/tasks/tasks.component.css'],
  templateUrl: 'app/tasks/tasks.component.html'
})
export default class TasksComponent implements OnInit {
  today: Date;
  tasks: Task[];
  queuedPomodoros: number;
  queueHeaderMapping: any;
  timerMinutes: number;

  constructor(
    private taskService: TaskService,
    private settingsService: SettingsService) {

    this.tasks = this.taskService.taskStore;
    this.today = new Date();
    this.queueHeaderMapping = settingsService.pluralsMap.tasks;
    this.timerMinutes = settingsService.timerMinutes;
  }

  ngOnInit(): void {
    this.updateQueuedPomodoros();
  }

  toggleTask(task: Task): void {
    task.queued = !task.queued;
    this.updateQueuedPomodoros();
  }

  private updateQueuedPomodoros(): void {
    this.queuedPomodoros = this.tasks
      .filter((Task: Task) => Task.queued)
      .reduce((pomodoros: number, queuedTask: Task) => {
      return pomodoros + queuedTask.pomodorosRequired;
    }, 0);
  }
};
```

Several aspects of the TasksComponent implementation are worth highlighting:

- We import the component's required child component and directives relatively. If we tried to bring them from the facade, we would get an undefined value because of a circular reference.
- We do not import all the required pipes, just its SHARED_PIPES alias token, registering it in the pipe's component decorator property.
- We inject the TaskService and SettingsService providers in the component, leveraging Angular's DI system. The dependencies are injected with accessors right from the constructor, becoming private class members on the spot.
- The tasks dataset and the pomodoro time duration are then populated from the bound services.

Our last step is to expose the facade required for this feature context. In all fairness, we are not meant to export everything on every context. We can simply get away with exporting only the main context component, while leaving any other sub component or directive outside the scope of the context facade. That is actually what we will do in here, since it is quite unlikely that any host component would ever need to include in its own view the `TaskIconsComponent`. We will nevertheless export the `TaskTooltipDirective`, since its implementation might be reused in the future by some other component dealing with the `[task]` input properties.

app/tasks/tasks.ts

```
import TasksComponent from './tasks.component';
import TaskTooltipDirective from './task-tooltip.directive';

const TASKS_DIRECTIVES: any[] = [
  TasksComponent,
  TaskTooltipDirective
];

export {
  TASKS_DIRECTIVES,
  TasksComponent,
  TaskTooltipDirective
};
```

Please check how we conform to the naming convention for alias tokens when grouping components and directives in their own alias token.

Defining the top root component

With all our feature contexts ready, time has come to define the top root component, which will kickstart the whole application as a cluster of components laid out in a tree hierarchy. The root component usually has a minimum implementation. Basically, its goal is to register the dependency providers the application will require as singletons at different levels of the component hierarchy, and instantiate in its view template the main child components that will eventually evolve into branches of child components.

app/app.component.ts

```
import { Component } from '@angular/core';
import { TIMER_DIRECTIVES } from './timer/timer';
import { TASKS_DIRECTIVES } from './tasks/tasks';
import { SHARED_PROVIDERS } from './shared/shared';

@Component({
  selector: 'pomodoro-app',
  directives: [TIMER_DIRECTIVES, TASKS_DIRECTIVES],
  providers: [SHARED_PROVIDERS],
  template: `
    <nav class="navbar navbar-default navbar-static-top">
      <div class="container">
        <div class="navbar-header">
```

```
            <strong class="navbar-brand">My Pomodoro App</strong>
          </div>
        </div>
      </nav>

      <pomodoro-timer-widget></pomodoro-timer-widget>
      <pomodoro-tasks></pomodoro-tasks>
      `
})
export default class AppComponent {}
```

Please check how we conveniently import the alias tokens and register them in the `providers` and `directives` properties of the component decorator. The `SHARED_PROVIDERS` token deserves a special mention. All the providers grouped by it are now available down the tree of components that hangs from this top root component, so there's no need to register it again and the state of each provider will remain consistent across the application domain.

The only exception to this would be to have one component at some level registering, for argument's sake, the `TaskService` as a provider. That would turn into a new instance of the service at that component level and for all its child components as well.

Bootstrapping the application

We now have a full-blown application featuring different functionality contexts, wrapped by a top root component. The last step of our endeavor will be to bootstrap the application, by importing the main top root component and passing it over to the `bootstrap()` function:

app/main.ts

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';

bootstrap(AppComponent);
```

This file must be imported from the main `index.html` file in order to trigger the whole process, by using a standard module loader such as SystemJS or WebPack. In the book repository, we use SystemJS so please refer to the chapter code there for further reference. In the `index.html` file, we will expect a custom element matching the top root component selector, as shown in the following `index.html` transcription:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My Angular 2 Pomodoro Application</title>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
    <script>
      System.import ('built/app/main')
      .then(null, console.error.bind(console));
    </script>
    <link rel="stylesheet"
href="node_modules/bootstrap/dist/css/bootstrap.min.css">

    <base href="/">
  </head>

  <body>
    <pomodoro-app>Loading...</pomodoro-app>
  </body>

</html>
```

With all the files in place, we can safely compile the project and see the results served by a web server in a browser window. If you have downloaded `package.json` (and related JSON files) from the book repo, please run `npm start` from your terminal window and enjoy. You

made a fantastic pomodoro application by yourself!

Summary

This chapter has definitely set the foundation for all the great applications that you will be building on top of Angular 2 from now on. The Angular 2 dependency management implementation is in fact one of the gems of this framework and a time saver. Application architectures based on component trees are not rocket science anymore, and we have followed this pattern to some extent while building web software in other frameworks such as Angular 1.x and React.

This chapter concludes our trip through the core of Angular 2 and its application architecture, setting up the standards that we will follow from now on while building applications on top of this new and exciting framework.

In the next chapters, we will focus on very specific tools and modules that we can use to solve everyday problems when crafting our web projects. We will see how to develop better HTTP networking clients with Angular 2.

Chapter 6. Asynchronous Data Services with Angular 2

Connecting to data services and APIs and handling asynchronous information is a common task in our everyday life as developers. In this sense, Angular 2 provides an unparalleled tool set to help its enthusiastic developers when it comes to consuming, digesting, and transforming all kinds of data fetched from data services.

There are so many possibilities that it would require an entire book to describe all that you can do to connect to APIs or consume information from the filesystem asynchronously through HTTP. In this book, we will only scratch the surface, but the insights covered in this chapter about the HTTP API and its companion classes and tools will give you all that you need to connect your applications to HTTP services in no time, leaving to your creativity all that you can do with them.

In this chapter, we will:

- Look at the different strategies for handling asynchronous data
- Introduce Observables and Observers
- Discuss functional reactive programming and RxJS
- Review the `HTTP` class and its API, as part of the
- Cover the `HTTP_PROVIDERS` module
- See all of the preceding points in action through actual code examples

Strategies for handling asynchronous information

Consuming information from an API is a common operation in our daily practice. We consume information over HTTP all the time—when authenticating users by sending out credentials to an authentication service, or when fetching the latest tweets in our favorite Twitter widget. Modern mobile devices have introduced an unparalleled way of consuming remote services, deferring the requests and response consumption until mobile connectivity is available. Responsivity and availability have become big deals. And although modern Internet connections are ultra-fast, there is always a response time involved when serving such information that forces us to put in place mechanisms to handle state in our applications in a transparent way for the end user.

This is not specific to scenarios where we need to consume information from an external resource. Sometimes, we might need to build functionalities that depend on time as a parameter of something, and we need to introduce code patterns that handle this deferred change in the application state.

For all these scenarios, we have always used code patterns, such as the callback pattern, where the function that triggers the asynchronous action expects another function in its signature, which will emit a sort of notification as soon as the asynchronous operation is completed:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}

function asynchronousOperation(callback) {
  setTimeout(callback, 5000);
}

asynchronousOperation(notifyCompletion);
```

The problem with this pattern is that code can become quite confusing and cumbersome as the application grows and more and more nested callbacks are introduced. In order to avoid this scenario, `Promises` introduced a new way of envisioning asynchronous data management by conforming to a neater and more solid interface, in which different asynchronous operations can be chained at the same level and even be split and returned from other functions:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}

function asynchronousOperation() {
  var promise = new Promise((resolve, reject) => {
    setTimeout(resolve, 5000);
  });
  return promise;
```

```
}
```

```
asynchronousOperation().then(notifyCompletion);
```

The preceding code example is perhaps a bit more verbose, but it definitely produces a more expressive and elegant interface for our `asynchronousOperation` function.

So, `Promises` took over the coding arena by storm and no developer out there seems to question the great value they bring to the game. So why do we need another paradigm? Well, because sometimes we might need to produce a response output that follows a more complex digest process as it is being returned, or even cancel the whole process. This cannot be done with `Promises`, because they are triggered as soon as they're instanced. 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 certain scenarios. `Promises` only 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 a one-time operation. 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 async data handling. This is where Observables come into the game.

Observables in a nutshell

An Observable is basically an async event emitter that informs another element, called the Observer, the state has changed. In order to do so, the Observable implements all of the machinery that it needs to produce and emit such async events, and it can be fired and canceled at any time regardless of whether it has emitted the expected data events already or not.

This pattern allows concurrent operations and more advanced logic, since the Observers that subscribe to the Observable async events will react to reflect the state change of the Observable they subscribe to.

These subscribers, which are the Observers we mentioned earlier, will keep listening to whatever happens in the Observable until the Observable is disposed, if that ever happens eventually. In the meantime, information will be updated throughout the application with no intervention whatsoever of triggering routines.

We can probably see all this with more transparency in an actual example. Let's refactor the example we covered when assessing promise-based async operations and replace the `setTimeout` command with `setInterval`:

```
function notifyCompletion() {
  console.log('Our asynchronous operation has been completed');
}

function asynchronousOperation() {
  var promise = new Promise((resolve, reject) => {
    setInterval(resolve, 2000);
  });
  return promise;
}

asynchronousOperation().then(notifyCompletion);
```

Copy and paste the preceding snippet in your browser's console window and see what happens: the text **Our asynchronous operation has been completed** will show up at the dev tools' console only once after 2 seconds and will never be rendered again. The `promise` resolved itself and the entire async event was terminated at that very moment.

Now, point your browser to an online JavaScript code playground such as JSBIN (https://jsbin.com/), and create a new code snippet enabling just the JavaScript and the **Console** tabs. Then, make sure you add the **RxJS library** from the **Add library** option dropdown (we will need this library to create observables, but don't panic; we will cover this later in the chapter) and insert the following code snippet:

```
var observable = Rx.Observable.create(observer => {
  setInterval(() => {
    observer.onNext('My async operation');
  }, 2000);
});
```

```
observable.subscribe(response => console.log(response));
```
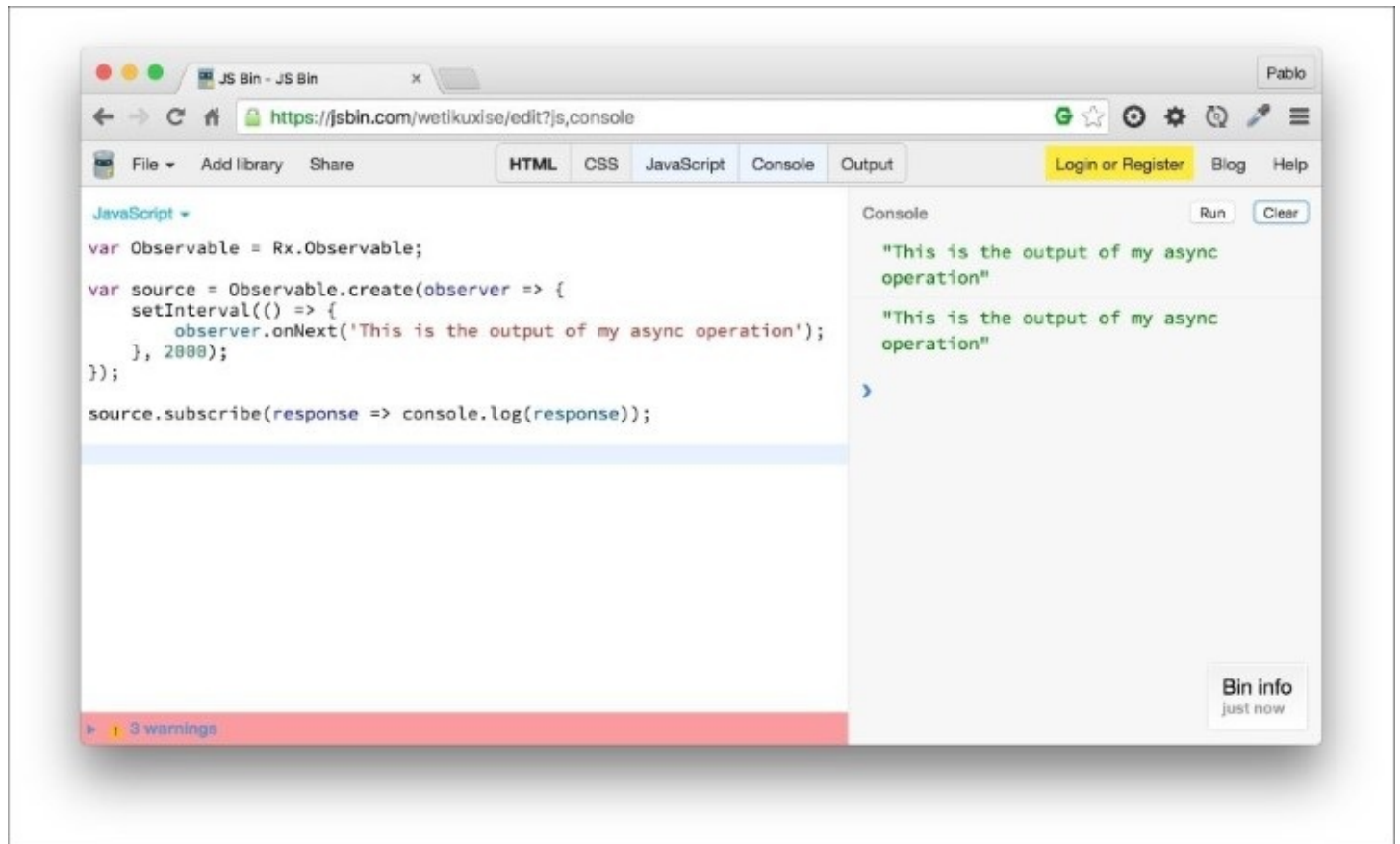

Run it and expect some message to appear on the right pane. 2 seconds later, we will see the same message showing up, and then again and again. In this simple example, we created an observable and then subscribed to its changes, throwing to the console whatever it emits (a simple message in this example) as a sort of push notification.

The Observable returns a stream of events and our subscribers receive prompt notification of those streamed events, acting accordingly. This is what the magic of Observables relies on— Observables do not perform an async operation and die (although we can configure them to do so), but start a stream of continuous events we can subscribe our subscribers to.

If we comment out the last line, nothing will happen. The **Console** pane will remain silent and all the magic will begin only when we subscribe our source object.

That's not all, however. This stream can be the subject of many operations before hitting the Observers subscribed to them. Just as we can grab a collection object, such as an array, and apply functional methods over it such as `map()` or `filter()` in order to transform and play around with the array items, we can do the same with the stream of events that are emitted by our Observables. This is what is known as reactive functional programming, and Angular 2 makes the most of this paradigm to handle asynchronous information.

Reactive functional programming in Angular 2

The Observable pattern stands at the core of what we know as reactive functional programming. Basically, the most basic implementation of a reactive functional script encompasses several concepts that we need to become familiar with:

- An Observable
- An Observer
- A timeline
- A stream of events featuring the same behavior as an objects collection
- A set of composable operators, also known as *Reactive Extensions*

Sounds daunting? It's not. Believe us when we tell you that all of the code you have gone through so far is much more complex than this. The big challenge here is to change your mindset and learn to think in a reactive fashion, and that is the main goal of this section.

If we want to put it simply, we can just say that reactive programming entails applying asynchronous subscriptions and transformations to Observable streams of events. We can imagine your poker face now, so let's put together a more descriptive example.

Think about an interaction device such as a keyboard. A keyboard has keys that the user presses. Each one of those key strokes triggers a key press event. That key press 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. Congratulations! You have an Observable sequence in the works here: the keyboard assumes the role of an Observable and emits a sequence of events that inform of the keys pressed no matter what happens next. Each keystroke event is agnostic from the rest and they follow a sequence along time. Now we mention events, one occurring after another. If we grab a group of elements of a given type (not necessarily the same type though) and wrap them in an object (which is what that timeline is), isn't it a collection? Moreover, that collection is spanned along time. So isn't that a stream? Indeed it is, so congratulations! You now have an event stream emitted by the Observable for your listening delight:

Keyboard keystrokes stream

TIME

Error event

These dots indicate keystroke events, ordered in time and representing different keystroke types

The vertical line indicates the stream has completed
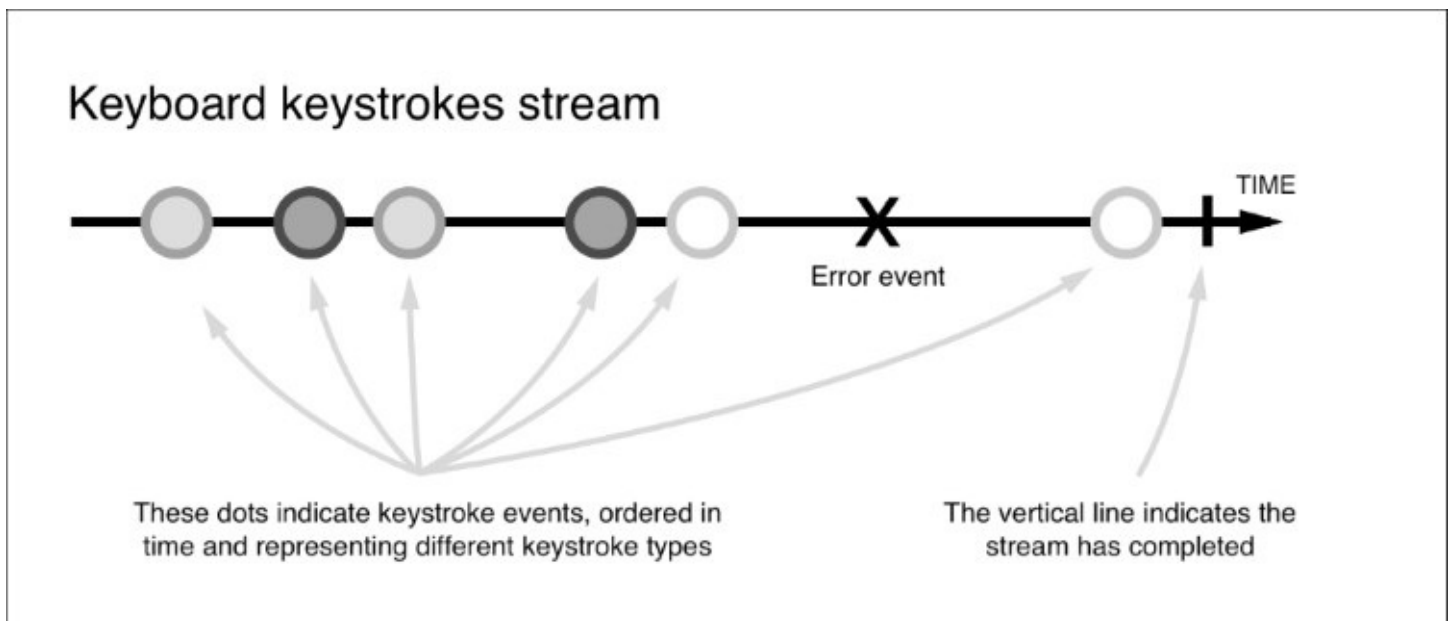
Our stream of key press events is looking really good, but we are getting notifications for all the keys pressed. What if we want to subscribe to a subset of the stream that observes the events represented by cursor keys? We could be coding a simple game and those are the keys that control our avatar in the game, while other key events are completely useless for the purpose of the game. Our ideal stream should contain only cursor keys' events then. This is where the functional part of the reactive paradigm comes into play.

Let's imagine we are grabbing that event stream and filtering it to include only the key press events that pertain to interactions with the cursor keys, disregarding all the others. On top of that, we are going to throttle the stream to allow a key press event to pass through only every 500 milliseconds, so we do not overflow whatever is listening at the other end of the wire. Now we have a new event stream, which is the by-product of the previous one, but represents only a subset of the information with a very specific goal:

Keyboard keystrokes stream

TIME

.filter(keyType === cursorKeyType)

We generate a new stream out from the first one by filtering cursor keys only.

Cursor keys keystrokes stream
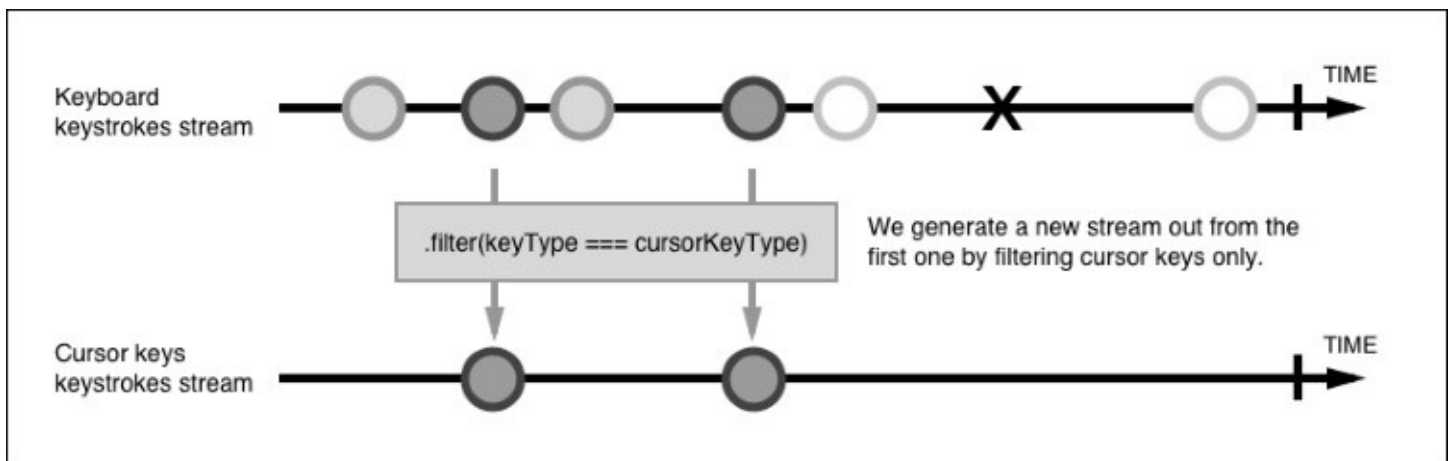
TIME

If we subscribe to this last stream, we can take action on each cursor key press and apply some gamification logic to whatever game we are developing at this time. We can even hook up several subscribers to that stream, or apply further transformations using other Reactive Extensions to create brand new streams that other subscribers can subscribe to in order to perform other business or presentation logic not related whatsoever to that performed by the original subscriber.

This logic can be ported to the realm of components to handle interaction events, asynchronous behavior, or (as this chapter aims to describe) the consumption and digestion of information served by an API service or data store. In order to do so, Angular 2 relies on RxJS, whose core module is required as a peer dependency when installing Angular 2. The RxJS API provides all that we need to put the aforementioned things to work on our asynchronous operations with Angular 2.

The RxJS library

As mentioned previously, Angular 2 comes with a peer dependency on RxJS, the JavaScript flavor of the `ReactiveX` library that allows us to create Observables and Observable sequences out of a large variety of scenarios, such as interaction events, `promises`, or callback functions, just to name a few. In that 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 comes with built-in support for a wide range of composable operators to transform, filter, and combine the resulting event streams. Its API provides convenient methods to subscribe Observers to these streams so that our scripts and components can respond accordingly to state changes or interaction inputs. While its API is so massive that covering it in detail is out of the scope of this book, we will highlight some bits of its most basic implementation in order for you to better understand how HTTP connections are handled by Angular 2.

Before jumping onto the HTTP API provided by Angular 2, let's create a simple example of an Observable event stream that we can transform with Reactive Extensions and subscribe observers to. To do so, let's pick the scenario described in the previous section.

We envisioned how a user interacting with our application through the keyboard can't turn into a timeline of keystrokes and, therefore, an event stream. Go back to JSBIN, delete the contents of the JavaScript pane, and then write down the following snippet:

```
var keyboardStream = Rx.Observable
  .fromEvent(document, 'keyup')
  .map(x => x.which);
```

The preceding code is pretty self-descriptive. We leverage the `Rx.Observable` class and its `fromEvent` method to create an event emitter that streams the `keyup` events that take place in the scope of the document object. Each of the event objects emitted is a complex object, so we simplify the streamed objects by mapping the event stream onto a new stream that contains only the key codes pertaining to each keystroke. The `map` method is a Reactive Extension that features the same behavior as the JavaScript `map` functional method. This is why we usually refer to this code style as reactive functional programming.

All right, so now we have an event stream of numeric keystrokes, but we are only interested in observing those events that inform of hits on the cursor keys. We can build a new stream out of an existing stream by applying more Reactive Extensions. So, let's do it with `keyboardStream` by filtering such a stream and returning only those events that are related to cursor keys. We will also map those events to their text correspondence for the sake of clarity. Append the following chunk of code right below the previous snippet:

```
var cursorMovesStream = keyboardStream.filter(x => {
    return  x > 36 && x < 41;
```

```
  })
  .map(x => {
    var direction;
    switch(x) {
      case 37:
        direction = 'left';
        break;
      case 38:
        direction = 'up';
        break;
      case 39:
        direction = 'right';
        break;
      default:
        direction = 'down';
    }
    return direction;
  });
```

We could have done all of this in a single action by chaining the `filter` and `map` methods to
the `keyboardStream` Observable and then subscribing to its output, but it's generally a good
idea to separate concerns. By shaping our code in this way, we have a generic keyboard events
stream that we can reuse later on for something completely different. So, our application can
scale up while keeping the code footprint to a minimum.

Now that we have mentioned subscribers, let's subscribe to our cursor moves stream and
throw the move commands at the console. We type the following statement at the end of our
script, then clear the **Console** pane, and click on the **Output** tab so that we can have a
document available:

```
cursorMovesStream.subscribe(e => console.log(e));
```

Click anywhere on the **Output** pane to put the focus on it and start typing random keyboard
keys and cursor keys:
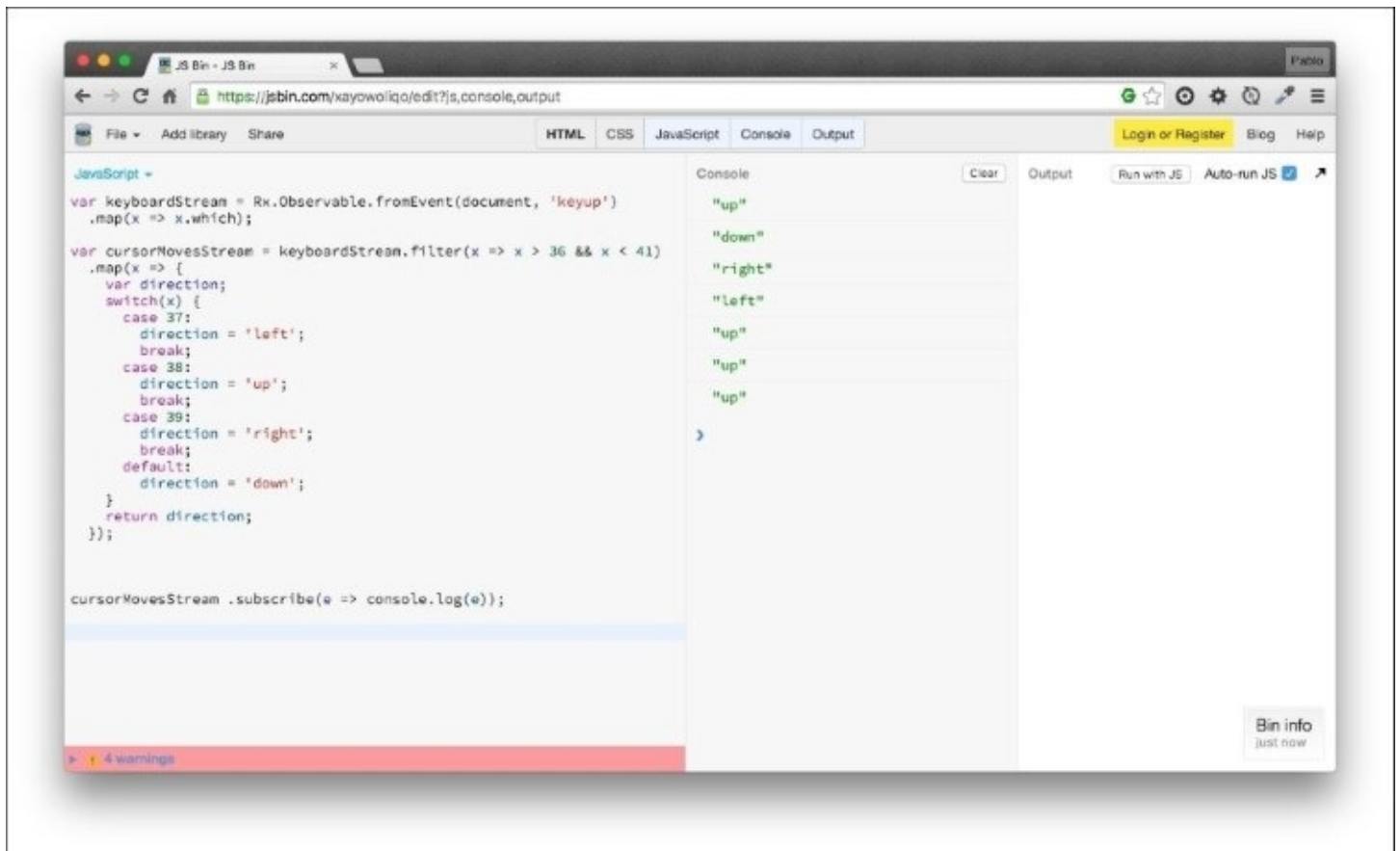
You are probably wondering how we can apply this pattern to an asynchronous scenario such as consuming information from a HTTP service. Basically, you have so far become used to submitting async requests to AJAX services and then delegating the response handling to a callback function or just piping it through a `promise`. Now, we will handle the call by returning an Observable. This Observable will emit the server response as an event in the context of a stream, which will be funneled through Reactive Extensions to better digest the response.

Introducing the HTTP API

The `Http` class provides a powerful API that abstracts all the operations required to handle asynchronous connections through a variety of HTTP methods, handling the responses in an easy and comfortable way. Its implementation has been made with a lot of care to ensure that programmers will feel at ease while developing solutions that take advantage of this class to connect to an API or a data resource.

In a nutshell, instances of the `Http` class (which has been implemented as an Injectable resource and can therefore be used in our classes constructors just by injecting it as a dependency provider) expose a connection method named `request()` to perform any type of `http` connection. The Angular 2 team has created some syntax shortcuts for the most common request operations, such as `GET`, `POST`, `PUT`, and every existing HTTP verb. So, creating an async `Http` request is as easy as this:

```
var requestOptions = new RequestOptions({
  method: RequestMethod.Get,
  url: '/my-api/my-data-store.json'
});
var request = new Request(requestOptions);
var myHttpRequest: Observable<Response> = http.request(request);
```

Also, all of this can be simplified into a single line of code:

```
var myHttpRequest: Observable<Response> = http.get('/my-api/my-data-store.json');
```

As we can see, the `Http` class connection methods operate by returning an Observable stream of `Response` object instances. This allows us to map and digest the output of the `Http` call as soon as it is available and subscribe Observers to the stream, which will process the information accordingly once it is returned, as many times as required:

```
myHttpRequest.map(response: Response => response.json())
  .subscribe(data => console.log(data));
```

In the preceding example, we map the responses emitted by the Observable event stream (as we can see, those are basically instances of the `Response` class provided by Angular 2) to a new collection of events representing JSON instances of each response. This is done by leveraging the `json()` method of the `Response` class, which we will cover later on in this chapter. Then we subscribe to the resulting stream and throw the output of the HTTP call to the **Console** once our Observer receives the data notification.

By doing this, we can respawn the `Http` request as many times as we need, and the rest of our machinery will react accordingly. We can even merge the event stream represented by the `Http` call with other related calls, and compose more complex Observable streams and data threads. The possibilities are endless.

When to use the Request and RequestOptionsArgs classes

We mentioned the `Request` and `RequestOptions` classes while introducing the `Http` class at first. On a regular basis, you will not need to make use of these low-level classes, mostly because the shortcut methods provided by the `Http` class abstract the need to declare the HTTP verb in use (`GET`, `POST`, and so on) and the URL we want to consume. With this being said, you will sometimes want to introduce special HTTP headers in your requests or append query string parameters automatically to each request, for argument's sake. That is why these classes can become quite handy in certain scenarios. Think of a use case where you want to add an authentication token to each request in order to prevent unauthorized users from reading data from one of your API endpoints.

In the following example, we read an authentication token and append it as a header to our request to a data service. Contrary to our example, we will inject the options hash object straight into the `Request` constructor, skipping the step of creating a `RequestOptions` object instance. Angular 2 provides a wrapper class for defining custom headers as well, and we will take advantage of it in this scenario. Let's figure out that we do have an API that expects all requests to include a custom header named Authorization, attaching the `authToken` that we received when logging into the system, which was then persisted in the browser's local storage layer, for instance:

```
var authToken = window.localStorage.getItem('auth_token');
var headers = new Headers();
headers.append('Authorization', `Token ${authToken}`);

var request = new Request({
  method: RequestMethod.Get,
  url: '/my-api/my-secured-data-store.json',
  headers: headers
});

var authRequest: Observable<Response> = http.request(request);
```

Again, we would like to note that apart from this scenario, you will seldom need to create custom request configurations, unless you want to delegate the creation of request configurations in a factory class or method and reuse the same `Http` wrapper all the time. Angular 2 gives you all the flexibility to go as far as you wish when abstracting your applications.

The Response object

As we have already seen, the HTTP requests performed by the Http class return an observable stream of Response class instances. Similar to the Request object, you will rarely find yourself in need of instantiating this class. However, understanding the Response class interface is quite useful in order to understand the status of our request, handle connection errors, and properly digest the information returned in the stream.

In our first example, we mapped the content of the stream as a stream of JSON objects by executing the json() method. This method parses the response body as a JSON object, or raises an exception if such a body cannot be parsed. Besides this method, the Response class exposes the text() method, which will parse and return the response body as a plain string.

Let's figure this out: we have a component that exposes a string field named bio, which is rendered in the component's template. We might want to serve this bio property from a REST API. Hence, we could implement such functionality in a few lines of code, like this:

```
http.get('/api/bio')
  .map(res: Response => res.txt)
  .subscribe(bio => this.bio = bio);
```

The Response object exposes other minor methods and some interesting properties, such as the numeric status property, which informs of the status code returned by the server. The bytesLoaded and totalBytes numeric properties become quite useful when scaffolding preload notifiers in progress events. Special mention goes to the headers property, which returns an object based on the Headers class (https://fetch.spec.whatwg.org/#headers-class) of the Fetch API.

Coverage for all the methods and properties available in the Response class API is definitely beyond the scope of this book, as you will not be using those on a regular basis, but we encourage you to expand your knowledge on the subject by visiting the official documentation (https://angular.io/docs).

Handling errors when performing Http requests

Handling errors raised in our requests by inspecting the information returned in the `Response` object is actually quite simple. We just need to inspect the value of its Boolean property, which will return `false` if the HTTP status of the response falls somewhere out side of the 2xx range, clearly indicating that our request could not be accomplished successfully. We can double-check that by inspecting the status property to understand the error code or the type property, which can assume the following values: `basic`, `cors`, `default`, `error`, or `opaque`. Inspecting the response headers and the `statusText` property of the `Response` object will provide insightful information about the origin of the error.

All in all, we are not meant to inspect those properties on every response message we get. Angular 2 provides an Observable operator to catch errors, injecting in its signature the `Response` object we require to inspect the previous properties:

```
http.get('/api/bio')
   .map(res: Response => res.txt)
   .subscribe(bio => this.bio = bio)
   .catch(error: Response => console.error(error));
```

In a normal scenario, you would want to inspect more data rather than the `error` properties, aside from logging that information in a more solid exception tracking system.

Injecting the Http class and the HTTP_PROVIDERS modules symbol

The `Http` class can be injected in our own components and custom classes by leveraging Angular's unique dependency injection system. So, if we ever need to implement HTTP calls, we need to import the class and bind it as a dependency in the list of component or directive providers, like this:

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

@Component({
  selector: 'bio',
  providers: [Http],
  template: '<div>{{bio}}</div>'
})
class Biography {
  bio: string;
  constructor(http: Http) {
    http.get('/api/bio')
      .map((res: Response) => res.text())
      .subscribe((bio: string) => this.bio = bio);
  }
}
```

In the code provided, we just follow up with the `bio` example that we pointed out in the previous section. Note how we are importing the `Http` type and injecting it as a dependency through the providers `collection` property of the `Component` decorator.

Usually, we need to perform multiple HTTP calls in different parts of our application, so it's usually recommended to include the `Http` class in the root injector rather than on a per component injector basis. To do so and keeping in mind that the `Http` class might require other providers such as the `RequestOptions` class, Angular 2 provides a set of injectable symbols wrapped inside the `HTTP_PROVIDERS` token.

We recommend that you use this set instead of the `Http` class to inject the dependencies required to perform HTTP requests throughout your application. For your convenience, it is advised to do so at the root component providers property, so its injector will make the provider available throughout its child components tree. Taking our pomodoro app as an example, we would need to import it at the `AppComponent` code unit and inject it into the component providers right away:

app/app.component.ts

```
import { Component } from '@angular/http';
import { TIMER_DIRECTIVES } from './timer/timer';
import { TASKS_DIRECTIVES } from './tasks/tasks';
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
```

```
@Component({
  selector: 'pomodoro-app',
  directives: [TIMER_DIRECTIVES, TASKS_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS],
  ...
})
export default class AppComponent {}
```

A real case study – serving Observable data through HTTP

In the previous chapter, we refactored our entire app into models, services, pipes, directives, and component files. One of those services was precisely the `TaskService` class, which is the bread and butter of our app, since it delivers the data that we need to build our task list and other related components.

In our example, the `TaskService` class was contained within the information we wanted to deliver. In a real-world scenario, you need to fetch that information from a server API or backend service. Let's update our example to emulate this scenario. First, we will remove the task information from the `TaskService` class and wrap it into an actual JSON file. Let's create a new JSON file inside the shared folder and populate it with the task information that we had hardcoded in the original `TaskService.ts` file, now in JSON format, though:

app/shared/data/raw-tasks.json

```
[{
    "name": "Code an HTML Table",
    "deadline": "Jun 23 2015",
    "pomodorosRequired": 1
}, {
    "name": "Sketch a wireframe for the new homepage",
    "deadline": "Jun 24 2016",
    "pomodorosRequired": 2
}, {
    "name": "Style table with Bootstrap styles",
    "deadline": "Jun 25 2016",
    "pomodorosRequired": 1
}, {
    "name": "Reinforce SEO with custom sitemap.xml",
    "deadline": "Jun 26 2016",
    "pomodorosRequired": 3
}]
```

With the data properly wrapped in its own file, we can consume it as if it were an actual backend service from our `TaskService` client class. However, we will need to conduct relevant changes in our `main.ts` file for that. The reason is that despite installing the `RxJS` bundle when installing all the Angular 2 peer dependencies, the reactive functional operators, like `map()`, do not become available straight away. We could import all of them at once by inserting the following line of code at some step at the beginning of our application initialization flow, such as the bootstrapping stage in `main.ts`:

```
import 'rxjs/Rx';
```

However, that would import all the reactive functional Operators, which will not be used at all and will consume an unnecessarily huge amount of bandwidth and resources. Instead, the

convention marks to import only what is needed, so append the following import line at the top of the `main.ts` file:

app/main.ts

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';

bootstrap(AppComponent, []);
```

When a reactive operator is imported this way, it gets automatically added to the `Observable` prototype, being then available for use throughout the entire application.

With all the dependencies properly in place, the time has come to refactor our `TaskService.ts` file. Open the service file and let's update the import statements block:

app/shared/services/task.service.ts

```
import { Injectable } from '@angular/code';
import { Task } from '../shared';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
```

First, we import in the `Http` and `Response` symbols so that we can annotate our objects later on. Remember anyway that the `HTTTP_PROVIDERS` token has already been injected at the top root component. The `Observable` symbol is imported from the `RxJS` library so that we can properly annotate the return types of our async `Http` requests.

Now we will replace the existing implementation with the following one. Basically, the `TaskService` modules evolve into a service with state that keeps exposing a `taskStore` property where we can fetch the tasks dataset. It also features an `Observable` property representing a task feed we can subscribe to in order to keep up to date of any new task that could be created in the future.

The constructor now features the `Http` dependency injected and bound as a private member to the `http` field:

app/shared/services/task.service.ts

```
@Injectable()
export default class TaskService {
  taskStore: Task[] = [];
  taskFeed: Observable<Task>;
  private taskObserver: any;
  private dataUrl = '/app/shared/data/raw-tasks.json';

  constructor(private http: Http) {
    this.taskFeed = new Observable(observer => {
      this.taskObserver = observer;
```

```
    });
    this.fetchTasks();
  }

  private fetchTasks(): void {
    this.http.get(this.dataUrl)
      .map(response => response.json())
      .map(stream => stream.map(res => {
        return {
          name: res.name,
          deadline: new Date(res.deadline),
          pomodorosRequired: res.pomodorosRequired,
          queued: res.queued
        }
      }))
      .subscribe(
        tasks => {
          this.taskStore = tasks;
          tasks.forEach(task => this.taskObserver.next(task))
        },
        error => console.log(error)
      );
  }
}
```

Let's take a minute to examine the new service implementation. We keep the `taskStore` property as usual, and we populate it once we fetch the whole graph of tasks available upon instantiating the service by calling the `fetchTasks()` private method from the constructor. But we have introduced an Observable member and an Observer field too.

Let's explain their role in detail. In the constructor, we initialize the Observable instance and also assign the Observable's built-in observer to our Observer member. This way, every time we want to notify a new task in the tasks Observable sequence to the service subscribers, we just need to proceed to execute the `next()` method of the Observer member. Therefore, as a singleton, the tasks service exposes a data store that is populated the moment it is instantiated for the first time, becoming such data fully available for all components that will eventually consume it.

On the other hand, external consumers of the service can subscribe to the `taskFeed` property and receive prompt notifications every time a new task is added to the sequence. We could spawn a web sockets client and leverage the Observer API to emit new tasks through our Observable sequence every time a new task is created on the server side. The components subscribing to that Observable sequence would receive the changes and reflect those changes in their state automatically with no additional logic required.

As we can see, this data handling pattern goes a step beyond the mere asynchronous data consumption flow we are used to with `promises` and callbacks, easing data updates and allowing for a better management of information offline.

Let's finish our implementation by updating the `TaskComponent` module. Since we are using

the same API we already had previously, the changes are minimal and are limited to tweaking the `ngOnInit` hook method:

app/tasks/tasks.component.ts

```
ngOnInit(): void {
  this.updateQueuedPomodoros();
  this.taskService.taskFeed.subscribe(newTask => {
    this.tasks.push(newTask);
    this.updateQueuedPomodoros();
  });
}
```

We need to subscribe to the tasks feed since Observables are cold. They are not initialized until some client actually subscribes to them, and the underlying Observer of the `taskFeed` Observable at `TaskService` is actually required within the internal subscription to the `http.get()` connection. Not subscribing to it would turn into an exception. With this change, we also ensured that our tasks table will remain up to date should a new task added to the overall tasks data store, without having to repopulate the whole tasks property. The call to `updateQueuedPomodoros()` is introduced in the callback as well should any new task be queued by default.

Now, execute the code and you will see the tasks seamlessly rendered on the table.

Adding tasks to our tasks service

Unfortunately, our code works in one way only now: we can consume data from the tasks JSON file but we cannot append new tasks if required. Ideally, we should be able to append our own tasks upon request and have the whole system reacting to these changes, so let's update our implementation for the `TaskService` class to insert and add a task method. Append the following method at the end of the service class:

app/shared/services/task.service.ts

```
addTask(task: Task): void {
  this.taskObserver.next(task);
}
```

This new method aligns with what we pointed out already about reactive service interfaces. Adding a new task object will turn into a new event published in the Observable events stream of tasks, so any active component which is already consuming the data graph and is subscribed to its changes will receive a prompt notification of the new task created and therefore can update its state accordingly.

Try it out yourself!

Tip

In all fairness, any new item request should be handled by means of a `POST` request and all the previous operations should be performed upon resolving the `Http.post()` request, like this:

```
const body = JSON.stringify(task);
const headers = new Headers();
headers.append('Content-Type', 'application/json');

addTask(task: Task): void {
    this.http.post(this.dataUrl, body, headers)
      .map(response => response.json())
      .subscribe((task: Task) =>
        this.taskObserver.next(task);
        }
      );
  }
```

Since server-side implementations are out of the scope of this book, we will leave it up to you to experiment with the `Http` module against RESTful APIs.

Summary

As we pointed out at the beginning of this chapter, it takes much more than a single chapter to cover in detail all the great things that can be done with the Angular 2 HTTP connection functionalities, but the good news is that we have covered pretty much all the tools and classes we need to do so.

The rest is just left to your imagination, so feel free to go the extra mile and put all of this knowledge into practice by creating brand new Twitter reader clients, newsfeed widgets, or blog engines, and assembling all kinds of components of your choice. 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 Functional extensions and the tiny but powerful `Http` class.

As we have already highlighted, the sky is the limit. But we still have a long and exciting way ahead. Now that we know how to consume asynchronous data in our components, let's discover how we can provide a broader user experience in our applications by routing users into different components. We will cover this in the next chapter.

Chapter 7. Routing in Angular 2

In the previous chapters, we did a great job separating concerns in our applications and adding different layers of abstraction to increase the maintainability in our Pomodoro app. However, we have neglected the visual side of things and the user experience part.

At this moment, our UI is bloated with components and stuff scattered across a single screen, and we need to provide a better navigational experience and a logical way to change the application's state intuitively.

This is the moment where routing acquires special relevance and gives us the opportunity to build a navigational narrative for our applications, allowing us to split the different areas of interest into different pages that are interconnected by a grid of links and URLs.

However, our application is only a set of components, so how do we deploy a navigation scheme between them? The Angular 2 router was built with componentization in mind. We will see how can we create our custom links and make components react to them in the following pages. In this chapter, we will:

- Discover how to define routes to switch components on and off and redirect them to other routes
- Trigger routes and load components in our views depending on the requested route
- Pass parameters to our components straight from our routes
- Look at the different component lifecycle hooks based on the routing stages
- Define different URL representation strategies

Adding support for the Angular 2 router

Same as we did when overviewing the Http directives and providers, all the types and tokens required for implementing routing support in our applications come from its own specific barrel. This barrel was already installed and configured back in [Chapter 1](#), *Creating Our Very First Component in Angular 2*, although we found two barrels related to routing in our installation and further configuration: `@angular/router` and `@angular/router-deprecated`. This is because the Angular team introduced a revamped routing mechanism when switching versions from Beta to Release Candidate. This new routing machinery, which aims to replace the routing API that Angular had been implementing since its Alpha version, also introduced relevant breaking changes with its previous incarnation. In order to ensure that applications built on top of the previous router could upgrade to Angular 2 Release Candidate seamlessly and prevent major issues, the Angular team made available a snapshot of the Beta Router, available from the `@angular/router-deprecated` barrel. That is why we installed and configured two routing packages.

Tip

At the time of closing the writing of this book, the new Angular 2 Router is still in a very early stage and lacks support for several functionalities that are commonly used in our web applications on a daily basis. That is way this chapter will focus on developing applications on top of the deprecated router yet we will highlight the differences between its API and the newer Angular 2 Router whereas possible. All in all the differences are minimal and learning how to use the deprecated router interface will become priceless for getting up to speed with the new router once it becomes final. Please refer to the book code repository to check the latest version of the code.

We also need to inform Angular about the base path we want to use, so it can properly build and recognize the URLs as the user browses the website, as we will see in the next section. Our first task will be to insert a base `href` statement within our `<HEAD>` element. Append the following line of code at the end of your code statement inside the `<head>` tag:

index.html

```
<base href="/">
```

The `base` tag informs the browser about the path it should follow while attempting to load external resources, such as media or CSS files, once it goes deeper into the URL hierarchy.

Now, we can start playing around with all the goodies existing in the router library. Prior to this, we would need to inform the dependency injector about how it can instantiate the tokens we will require later on while implementing the routing features in our components. All these providers are accessible from the `ROUTER_PROVIDERS` symbol. In a similar fashion as we did with `HTTP_PROVIDERS`, we need to declare it in the providers property of the top root component so that it is available for all its child components' injectors.

Open your top component module and append the following `import` statement to the existing block of imported symbols. Then, add it to the `providers` property of the component decorator:

app/app.component.ts

```
...
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import { ROUTER_PROVIDERS } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  template: `
...
```

Setting up the router service

With the providers and directives in place, our first step will be to turn our main host component into a router component. Basically, any component can become a routing component just by conforming to the following requirements:

- Just like the component class is flagged with a `@Component` decorator, we want to decorate it with a `@RouteConfig` decorator.
- The `@RouteConfig` decorator is configured with an array of `RouteDefinitions`, which are basically object literals defining a path identified with a name and pointing to a component type.
- The component decorated with the `@RouteConfig` decorator is then supposed to include a `RouterOutlet` directive in its template. This element will become the placeholder where the components will be loaded and rendered upon loading a route pointing to each of them, removing any previous component existing there, if any.

In this sense, it is right to say that the router watches for state changes in the browser URL and then searches for a `RouteDefinition` object whose path property matches the existing URL. Then, it instantiates the component defined in such route definition inside the placeholder represented by the router outlet directive, which is meant to live in the template belonging to the component decorated with that router configuration.

Let's see all this through a real example. As we mentioned while introducing this chapter, our application needs a better navigation architecture in order to be more usable and intuitive. After splitting all our logic into different components in Chapter 5, *Building an Application with Angular 2 Components*, we will define different routes to use each one, implementing the following logic:

- The user reaches our app and checks the current listing of the tasks pending to be done. The user can schedule the tasks to be done in order to get the required time estimation for the next Pomodoro session.
- If desired, the user can jump onto another page and see a create task form (we will create the form, but will not implement its editing features until the next chapter).
- The user can choose any task at any time and begin the Pomodoro session required to accomplish it.
- The user can move back and forth across the pages she or he has already visited.

Building a new component for demonstration purposes

So far, we have built two well-differentiated components we can leverage to deliver a multipage navigation. But in order to provide a better user experience, we might need a third one. We will now introduce the form component we will be elaborating more thoroughly in [Chapter 8](), *Forms and Authentication handling in Angular 2,* as a way to have more navigation options in our example.

We will create a component in our `tasks` feature folder, anticipating the form we will use in the next chapter to publish new tasks. Create the following files in the locations pointed out for each one:

app/tasks/task-editor.component.ts

```
import { Component } from 'angular2/core';
import { ROUTER_DIRECTIVES } from 'angular2/router';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  templateUrl: 'app/tasks/task-editor.component.html'
})
export default class TaskEditorComponent {
  constructor() {}
}
```

app/tasks/task-editor.component.html

```
<form class="container">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      required>
  </div>

  <div class="form-group">
    <input type="Date"
      class="form-control"
      required>
  </div>

  <div class="form-group">
    <input type="number"
      class="form-control"
      placeholder="Pomodoros required"
      min="1"
      max="4"
      required>
  </div>

  <div class="form-group">    <input type="checkbox" name="queued">
```

```
      <label for="queued"> this task by default?</label>
    </div>

    <p>
      <input type="submit" class="btn btn-success" value="Save">
      <a href="/" class="btn btn-danger">Cancel</a>
    </p>
</form>
```

This is the most basic definition of a component, but we will also bring the
ROUTER_DIRECTIVES symbol from the router library. This will provide us support, as we will
see later on, to include routing directives in our HTML template. This will be used to
introduce links in our template to jump to other components, as we will see shortly. Last but
not least, we need to expose this new component from our feature folder facade:

app/tasks/tasks.ts

```
import TasksComponent from './tasks.component';
import TaskEditorComponent from './task-editor.component';
import TaskTooltipDirective from './task-tooltip.directive';

const TASKS_DIRECTIVES: any[] = [
  TasksComponent,
  TaskEditorComponent,
  TaskTooltipDirective
];

export {
  TASKS_DIRECTIVES,
  TasksComponent,
  TaskEditorComponent,
  TaskTooltipDirective
};
```

Configuring the RouteConfig decorator with the RouteDefinition instances

In order to achieve these goals, we need to start building our top router, which will be in charge of kicking off the routes' scaffolding. The logical path begins in our top root component. Open its file module and import the following tokens, right next to the `ROUTER_PROVIDERS` symbol we imported at the beginning of this chapter. The code is as follows:

app/app.component.ts

```
...
import {
  ROUTER_PROVIDERS,
  RouteConfig,
  ROUTER_DIRECTIVES
} from '@angular/router-deprecated';
import { TimerComponent } from './timer/timer';
import {
  TasksComponent,
  TaskEditorComponent } from './tasks/tasks';
...
```

The `RouteConfig` represents the decorator type that will turn our component into a router component. The `ROUTER_DIRECTIVES` symbol wraps the view directives we will need shortly to link to these routes. We also import the tokens of all the three components we will be dealing with. As we will shortly see, each route needs to declare the type of the component we are routing the browser to.

Let's continue by replacing the directives in our `AppComponent` module by the `ROUTER_DIRECTIVES` symbol, since we will not need to declare the facade tokens of the components that lived in its template anymore. The router will handle this for us:

app/app.component.ts

```
@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  template: `
  ...
})
```

Now, let's expand the `component` class definition with the `RouteConfig` decorator by appending the following decorator right after the `@Component` decorator block and before the class statement:

app/app.component.ts

```
@RouteConfig([
  { path: '',
    name: 'TasksComponent',
    component: TasksComponent
  },
 {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
  }, {
    path: 'timer',
    name: 'TimerComponent',
    component: TimerComponent
  }
])
export default class AppComponent {}
```

As we pointed out at the beginning of this chapter, the `RouteConfig` decorator must be populated with an array of `RouteDefinition` objects, each one specifying a path that, once reached by the user, will enable the component whose type we have defined in the component property.

Tip

Note: The new Router replaced the `@RouteConfig` decorator by the `@Routes` decorator. The name property is removed from the route definitions schema and route matching is performed just by checking the path value.

In the previous example, our host component will react to three different routes and thus serve the `TasksComponent` item, the `TimerComponent` item, or the `TaskEditorComponent` item depending on the route's path.

Tip

Here we stumble upon another common convention in the previous version fo the Angular 2 router: naming routes with the same name as the component they refer to. As we will see shortly, we will use each route name for populating the links pointing to each resource, so naming routes after the component they will activate becomes pretty useful and intuitive when it comes to assessing the target of each link found in the template. This convention is no longer enforced in the new router, since only paths are used to perform route matching.

There are two questions at this point: where will these components be rendered and how will we trigger each route? In order to answer these questions, we need to look into the router directives in detail.

The router directives – RouterOutlet and RouterLink

The `ROUTER_DIRECTIVES` symbol gives us access to the only two directives we will need in our applications.

First, the `RouterOutlet` directive is the placeholder directive where the different components whose paths have been navigated to by the user will be rendered. The `RouterLink` is used as an attribute directive to help the HTML controls behave as anchors or link buttons leading to the different routes by specifying the unique name each route is configured with, as we will see next.

In the previous chapters, we configured our top root component template to display a cute `nav` bar header, followed by the custom elements representing the Pomodoro timer and the tasks list. Now, we will strip the HTML template out from the component decorator definition and save it into its own template file, in order to access it more conveniently when editing the HTML is required. Also, we will refactor it into a router-friendly component template with links pointing to the different views or states our application can feature, as follows:

app/app.component.ts

```
 ...
 @Component({
selector: 'pomodoro-app',
directives: [ROUTER_DIRECTIVES],
providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
templateUrl: 'app/app.component.html'
})
...
```

app/app.component.html

```html
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro App</strong>
    </div>
    <ul class="nav navbar-nav navbar-right">
      <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
      <li><a [routerLink]="['TimerComponent']">Timer</a></li>
      <li><a [routerLink]="['TaskEditorComponent']">
          Publish Task</a>
      </li>
    </ul>
  </div>
</nav>

<router-outlet></router-outlet>
```

As you can see in the template, the location where the components used to live has been replaced by the `<router-outlet>` directive, and three new links compound up our beautiful

nav bar. Reload the page and see how our tasks list is rendered on the screen, and then click on the **Timer** link. Awesome! The `TimerComponent` item just loaded on the screen. Click on the other link or hit back on your browser to see how you can jump across the different components seamlessly.

Let's take a look at these links more closely, taking the link pointing to the timer as an example:

```
<a [routerLink]="['TimerComponent']">Timer</a>
```

The morphology of the `routerLink` directive is pretty self-explanatory. On the right-hand side of the equal symbol, it expects an array of route names corresponding to the named route and, optionally, the subroutes within the former that we want to navigate to. Most of the time, we will just see one unique string value. However, the array can allocate many values depending on whether the component whose named path we are pointing to hosts its own router with named routes as well. In this case, the subroute names we want to load will be declared next in the strings array. This is why it is so convenient and recommended to name our routes after the components they point at.

Note

The new Release Candidate Router deprecates named routes, favoring URL paths instead. Therefore, the `[routerLink]` directive will expect a full path as a value.

The `routerLink` definition also leaves room to add parameters declaratively, so we can trigger dynamic routes at runtime. We will tap into all these features throughout the next sections, but now we need to answer one important question. What if we want to navigate to a component imperatively without actually clicking on a link, but rather as the by-product of an action performed within the component's controller class?

Triggering routes imperatively

Perhaps, you would like to jump on our timer by selecting the task we want to work on. We have already set up a behavior that displayed a queued label whenever each task was picked for being done. We will leverage the same behavior to create a work on button that will redirect the user to the timer component.

First, let's inject the `Router` type as a dependency in our `TasksComponent` module, so we can gain access imperatively to its methods. Since we already declared the `ROUTER_PROVIDERS` symbol while bootstrapping the application, Angular 2 will take care of injecting the router type if properly declared in our component, so let's do it. Add the following import statement at the top of the component, right after the existing ones:

app/tasks/tasks.component.ts

```
import { Router } from '@angular/router-deprecated';
```

Now, let's update the constructor to inject the router type. We will mark the constructor as a private form so that it becomes privately available from the component members:

app/tasks/tasks.component.ts

```
...
constructor(
  private taskService: TaskService,
  private settingsService: SettingsService,
  private router: Router) {
    this.tasks = this.taskService.taskStore;
    this.today = new Date();
    this.queueHeaderMapping = settingsService.pluralsMap.tasks;
    this.timerMinutes = settingsService.timerMinutes;
  }
...
```

Now, let's add a method right below the `updateQueuedPomodoros()` method, which will lead the user imperatively to the task route upon executing it:

```
workOn(): void {
  this.router.navigate(['TimerComponent']);
}
```

How do we execute it? Let's introduce a new button in our table, next to the toggle task button at the last cell on each row, with a click handler pointing to the preceding method. The code is as follows:

app/tasks/tasks.component.html

```
<td>
  <button type="button" class="btn btn-default btn-xs"
    [ngSwitch]="task.queued"
```

```
  (click)="toggleTask(task)">
  <template [ngSwitchWhen]="false">
    <i class="glyphicon glyphicon-plus-sign"></i>
    Add
  </template>
  <template [ngSwitchWhen]="true">
    <i class="glyphicon glyphicon-minus-sign"></i>
    Remove
  </template>
  <template ngSwitchDefault>
    <i class="glyphicon glyphicon-plus-sign"></i>
    Add
  </template>
  </button>
  <button type="button"
    class="btn btn-default btn-xs"
    *ngIf="task.queued"
    (click)="workOn()">
    <i class="glyphicon glyphicon-expand"></i> Start
  </button>
</td>
```

A convenient `NgIf` directive will display the button only when required. We have included an icon for cosmetic purposes, but feel free to remove it or replace it by any other glyph of your choice.

Now reload the table, set out any task to be done, and click on the button that appears. Voila! You will be redirected to the timer to begin working on that task if desired.

Note

The `navigate()` method of the new Router will expect a string containing the full path instead.

CSS hooks for active routes

We have seen how to turn any link or DOM element into a hyperlink pointing to a named route that instantiates a component. However, it would be great to provide some kind of visual cue about the active link at any given time. That is precisely one of the side features implemented in the `routerLink` directive: whenever the route defined on a `routerLink` directive becomes active (regardless of whether the user reached that route declaratively or imperatively), the element will be decorated with the router-link-active class. We can therefore introduce specific CSS definitions in our components' style sheets to highlight if a specific link is active or not.

We will see all this functionality in action just by tweaking our top parent component a bit. Open the top root component controller class file and insert the following style sheet in `hte` component decorator configuration:

app/app.component.ts

```
@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES],
  providers: [SHARED_PROVIDERS, HTTP_PROVIDERS, ROUTER_PROVIDERS],
  styles: [`
      .router-link-active {
          font-weight: bold;
          border-bottom: 2px #d9534f solid;
      }
  `],
  templateUrl: 'app/app.component.html'
})
```

Now reload the browser and click on any link or navigate to a task timer from the tasks table, keeping an eye on the visual state of the top nav bar. The active link will be properly enhanced with the styling we defined. If you inspect the code, you will see the router-link-active class rendered on the active link each time.

Handling route parameters

We have configured pretty basic paths in our routes so far, but what if we want to build dynamic paths with support for parameters or values created at runtime? Creating (and navigating to) URLs that load specific items from our data stores is a common action we need to confront on a daily basis. For instance, we might need to provide a master-detail browsing functionality, so each generated URL living in the master page contains the identifiers required to load each item once the user reaches the detail page.

We are basically tackling a double trouble here: creating URLs with dynamic parameters at runtime and parsing the value of such parameters. No problem, the Angular router has got our back and we will see how using a real example.

Passing dynamic parameters in our routes

We updated the tasks list to display a button leading to the timer component page when clicked. But we just load the timer component with no context whatsoever of what task we are supposed to work on once we get there. Let's extend the component to display the task we picked prior to jumping to this page.

First, let's get back to the tasks list component template and update the signature of the button that triggers the navigation to the timer component in order to include the index of the task item corresponding to that loop iteration:

app/tasks/tasks.component.html

```
...
<button type="button"
    class="btn btn-default btn-xs"
    *ngIf="task.queued"
    (click)="workOn(i)">
    <i class="glyphicon glyphicon-expand"></i> Start
  </button>
...
```

Remember that such an index was generated at every iteration of the `NgFor` directive that rendered the table rows. Now that the call incorporates the index in its signature, we just need to modify the payload of the navigate method:

```
workOn(index: number): void {
  this.router.navigate(['Timer', { id: index }]);
}
```

If this had been a `routerLink` directive, the parameters would have been defined in the same way: a hash object following the path name string (or strings, as we will see while tapping into the child routers) inside the array. This is the way parameters are added to the generated link. However, if we click on any button now, we will see that the dynamic ID values are appended as query string parameters. While this might suffice in some scenarios, we are after a more elegant workaround for this. So, let's update our route definition to include the parameter in the path. Go back to our top root component and update the route inside the `RouteConfig` decorator as follows:

app/app.component.ts

```
...
}, {
  path: 'timer/:id',
  name: 'TimerComponent',
  component: TimerComponent
}
...
```

Refresh the application, schedule the last task on the table, and click on the **Start** button. You will see how the browser loads the Timer component under a URL like `/timer/3`.

Each path can contain as many tokens prefixed by a colon as required. These tokens will be translated to the actual values when we act on a `routerLink` directive or execute the navigate method of the `Router` class by passing a hash of the key/value pairs, matching each token with its corresponding key. So, in a nutshell, we can define route paths as follows:

```
{
  path: '/products/:category/:id',
  name: 'ProductsByCategoryComponent',
  component: ProductsByCategoryComponent
}
```

Then, we can execute any given route such as the one depicted earlier as follows:

```
<a [routerLink]="['ProductsByCategoryComponent', {
  category: 'toys',
  id: 452
}]">See Toy</a>
```

The same applies to the routes called imperatively:

```
router.navigate(['ProductsByCategoryComponent', {
  category: 'toys',
  id: 452
}]);
```

Parsing route parameters with the RouteParams service

Great! Now, we are passing the index of the `task` item we want to work on loading the timer, but how do we parse that parameter from the URL? The Angular router provides a convenient injectable type (already included in `ROUTER_PROVIDERS`) named `RouteParams` that we can use from the components handled by the router to fetch the parameters defined in the route definition path.

Open our `timer` component and import it with the following `import` statement. Also, let's inject the `TaskService` provider, so we can retrieve information from the task item requested:

app/timer/timer-widget.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SettingsService, TaskService } from '../shared/shared';
import { RouteParams } from '@angular/router-deprecated';
...
```

We need to alter the component's definition in order to assign the `TaskService` as an annotated dependency for this component, so the injector can properly perform the provider lookup.

Note

The new Release Candidate router has deprecated the `RouteParams` class, favoring the new `RouteSegments` class, which exposes more and more useful methods and helpers. Please refer to the official documentation for broader insights on its API.

We will also leverage this action to insert the interpolated title corresponding to the requested task in the component template:

app/timer/timer-widget.component.ts

```
...
@Component({
  selector: 'pomodoro-timer-widget',
  template: `
    <div class="text-center">
      <img src="/app/shared/assets/img/pomodoro.png">
      <h3><small>{{ taskName }}</small></h3>
      <h1> {{ minutes }}:{{ seconds  | number: '2.0' }} </h1>
      <p>
        <button (click)="togglePause()" class="btn btn-danger">
        {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
        </button>
      </p>
    </div>`
})
...
```

The `taskName` variable is the placeholder we will be using to interpolate the name of the task.

With all this in place, let's update our constructor to bring both the `RouteParams` type and the `TaskService` classes to the game as private class members injected from the constructor:

app/timer/timer-widget.component.ts

```
...
constructor(
  private settingsService: SettingsService,
  private routeParams: RouteParams,
  private taskService: TaskService) {
    this.buttonLabelsMap = settingsService.labelsMap.timer;
}
...
```

With these types now available in our class, we can leverage the `ngOnInit` hook to fetch the task details of the item in the tasks array corresponding to the index passed as a parameter. Waiting for the `OnInit` stage is not easy, since we will find issues when trying to access the properties contained in `routeParams` before that stage:

app/timer/timer-widget.component.ts

```
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }
}
```

How do we fetch the value from that `id` parameter? The `RouteParams` object exposes a `get(param: string)` method we can use to address parameters by name. In our example, we retrieved the value of the `id` parameter by executing the `routeParams.get('id')` command in the `ngOnInit()` hook method. Basically, this is how we get parameter values from our routes. First, we grab an instance of the `RouteParams` class through the component injector and then we retrieve values by executing its getter function, which will expect a string parameter with the name of the token corresponding to the parameter we need.

Note

It is important to note that we are fetching the data already persisted in the `taskStore` property of our `TaskService` provider. Since it is a singleton, available throughout the entire application by means of the Angular DI machinery, which had been already populated at `TaskComponent`, all the information we require is already there. Things would become trickier if we load directly each timer URL. In those cases, the information would have not been fetched yet, so we would have to subscribe to the service in order to force it to load the data through its underlying Http client. We saw this in Chapter 6, *Asynchronous Data Services with Angular 2*; applying the async pipe to the `taskName` interpolation in the template would be

required. For the sake of simplicity, we will skip that refactoring here, but we encourage you to tweak the component to extend support for this scenario as well.

Defining child routers

As our applications scale, the idea of bundling all the route definitions in a centralized location (for example, the root component) does not seem like a good approach. The more routes we define there, the harder it will be to maintain the application, let alone the tight coupling we generate between our components (that are meant to be as much reusable and application-agnostic as possible) and the application itself.

This is why it is generally a good practice to split and wrap the route definitions that apply to a specific feature around a router configuration defined on a specific component per feature level, usually the root component that wraps that feature context. The Angular team had this idea in mind when the `Router` library was designed and thus implemented support to extend a route with children routes while keeping the parent route fully agnostic of what routes are defined above its layer of functionality.

Let's see all this through an actual example. We updated our timer recently to display the name of the task we wanted to work on after selecting it from the table. However, what if we want to keep providing a standalone timer not bound to any specific task? This way, we can leverage the countdown functionality for any impromptu task without having to create it beforehand.

So, we will want to give access to the timer in two flavors:

- `timer`: This will load the timer as it is without pointing to any specific task
- `timer/task/{id}`: This will load the timer specifying a task name, where `{id}` is the index of the task we want to load from the overall tasks array.

We will begin by updating the main root component, now turned into a router component, to turn the `/timer/:id` path into a path pointing to a child router component. Open the component and replace the route definition pointing to the timer component using the following definition:

app/app.component.ts

```
{
  path: 'timer/...',
  name: 'TimerComponent',
  component: TimerComponent
}
```

That's it. The ellipsis right next to the route path informs the Angular Router that it should expect route definitions nested within that component. The problem here is that a component should not route to itself, since it cannot instantiate itself inside its own `RouterOutlet` directive. This is why we need to proxy the timer component with a router component for this example. So, let's create a routing component for our timer inside its own folder for simplicity sake. A routing component is, by definition, a component with no implementation other that to serve as a component dispatcher depending on routes. Their implementation, if

any, is generally pretty limited, and it basically entails the `RouteConfig` decorator containing the routes delivered by that feature context and the `RouterOutlet` present in its template. Routing components are indeed a good way to decouple routing functionalities from the specific implementation of each component in the context of that feature, ensuring full reusability of its non-routing components of that feature.

Open the `timer` feature folder and create a file for our timer routing component with the following implementation:

app/timer/timer.component.ts

```
import { Component } from '@angular/core';
import { RouteConfig, ROUTER_DIRECTIVES } from '@angular/router-deprecated';
import TimerWidgetComponent from './timer-widget.component';

@Component({
  selector: 'pomodoro-timer',
  directives: [ROUTER_DIRECTIVES],
  template: '<router-outlet></router-outlet>'
})
@RouteConfig([
  { path: '/', name: 'GenericTimer',
    component: TimerWidgetComponent,
    useAsDefault: true }, {
    path: '/task/:id',
    name: 'TaskTimer',
    component: TimerWidgetComponent
  }
])
export default class TimerComponent {}
```

As we can see, we have created a component class with no implementation other than dispatching routes to the `TimerWidgetComponent` component itself. The `RouteConfig` type allows us to create a proper decorator containing route definitions and `ROUTER_DIRECTIVES` will allow us to bind the `RouterOutlet` directive in the component template.

Please pay attention to the new `useAsDefault` Boolean property in the first route definition. This informs the router that if no matching paths are found, the `Router` class should load this route by default.

Note

At the time of this writing, the new Release Candidate Router still does not implement the `useAsDefault` property, but it is planned to be implemented by its final version. Please refer to the official documentation for further details.

It is important to note that any component acting as a router component can have its own implementation living in parallel to the `RouterOutlet` directive. Just like we did with `AppComponent`, we can provide additional functionalities to our component other than the mere routing features.

All right, we have a component now that can redirect users to our timer component in two flavors, but how do we link to it?

Linking to child routes

There is no difference whatsoever in linking to a child router and linking to any given route managed by a top router, except for the fact that we will populate the route names array with the names of the child routes we want to link as well. Open the `PomodoroTaskList` component and refactor the `workOn()` method to look like this:

```
workOn(index: number): void {
    this.router.navigate(['TimerComponent', 'TaskTimer', {    id: index }]);
}
```

Here, we are telling Angular to link to the route named `TimerComponent` (hence the importance of naming our routes after each target component's name). Since this is a parent route (remember the ellipsis) configured at our top root router, we need to provide the name of the child route to load from within the routes configured at the child router level, `TaskTimer` in this case. Obviously, we will compound up the route with the ID information required for loading the task requested. Click on any task and see how the timer is loaded, displaying the task name we wish to work on.

This implementation approach gives the component the chance to be displayed with or without Task ID information. This way, we can keep on browsing to the timer functionality, either from the top nav link or by clicking the **Start** button at the tasks table.

Just remember we configured the main route definition with the `useAsDefault` property set as true, remember? This means that anything pointing to the timer route will degrade gracefully to this last route definition once we reach the child route domain.

The Router lifecycle hooks

Just like components go through a set of different phases during their lifetime, a routing operation goes through different lifecycle stages. Each one is accessible from a different lifecycle hook which, just like components, can be handled by implementing a specific interface in the component subject of the routing action. The only exception for this is the earliest hook in the routing lifecycle, the `CanActivate` hook, which takes the shape of a decorator annotation, since it is meant to be called before the component is even instantiated.

The CanActivate hook

The `CanActivate` hook, presented as a decorator annotating the component, is checked by the `Router` right before it is instantiated. It will need its setup to be configured with a function that is intended to return a Boolean value (or a `Promise`-typed Boolean value) indicating whether the component should be finally activated or not:

```
@CanActivate((next, prev) => boolean | Promise<boolean>)
```

The `@CanActivate` decorator is therefore a function that expects another function as an argument, expecting the latter two `ComponentInstruction` objects as parameters in its signature: the first argument represents the route we want to navigate to and the second argument represents the route we are coming from. These objects expose useful properties about the route we come from and the component we aim to instantiate: path, parameters, component type, and so on.

Note

This hook represents a good point in the overall component's lifecycle to implement behaviors such as session validation, allowing us to protect areas of our application. Unfortunately the `CanActivate` hook does not natively support dependency injection, which makes harder to introduce advanced business logic prior to activate a route. The next chapters will describe workarounds for scenarios such as user authentication.

In the following example, we password-protect the form so that it won't be instantiated should the user enters the wrong passphrase. First, open the `TaskEditorComponent` module file and import all that we will need for our first experiment, along with all the symbols required for implementing the interfaces for the routing lifecycle hooks we will see throughout this chapter. Then, proceed to apply the `CanActivate` decorator to the component class:

app/tasks/task-editor.component.ts

```
import { Component } from '@angular/core';
import {
  ROUTER_DIRECTIVES,
  CanActivate,
  ComponentInstruction,
  OnActivate,
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  templateUrl: 'app/tasks/task-editor.component.html'
})
@CanActivate((
  next: ComponentInstruction,
  prev: ComponentInstruction): boolean => {
```

```
      let passPhrase = prompt('Say the magic words');
      return (passPhrase === 'open sesame');
   }
)
export default class TaskEditorComponent {
...
```

As you can see, we are populating the `@CanActivate` decorator with an arrow function declaring two `ComponentInstruction` typed arguments (which are not actually required for our example, although they have been included here for instructional purposes). The arrow function returns a Boolean value depending on whether the user types the correct case-sensitive passphrase. We would advise you to inspect the next and previous parameters in the console to get yourself more acquainted with the information these two useful objects provide.

By the way, did you notice that we declared the `ROUTER_DIRECTIVES` token in the directives property? The routing directives are not required for our overview of the different routing lifecycle hooks, but now we are tweaking this component and will keep updating it to test drive the different lifecycle hooks. Let's introduce a convenient back button, leveraging the **Cancel** button already present in the component template:

app/tasks/task-editor.component.html

```
<form class="container">
...
  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TasksComponent']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

The OnActivate Hook

The `OnActivate` hook allows us to perform custom actions once the route navigation to the component has been successfully accomplished. We can easily handle it by implementing a simple interface. These custom actions can even encompass asynchronous operations, in which case, we just need to `return` a `Promise` from the interface function. If so, the route will only change once the promised has been resolved.

Let's see an actual example where we will introduce a new functionality by changing the title of our form page. To do so, we will keep working on the `TaskEditorComponent` module to bring support for the `OnActivate` hook interface and the `Title` class whose API exposes utility methods (https://angular.io/docs/ts/latest/api/platform/browser/Title-class.html) to set or get the page title while executing applications in web browsers. Let's import the `Title` symbol and declare it as a provider in the component to make it available for the injector (you can also inject it earlier at the top root component should you wish to interact with this object in other components):

app/tasks/task-editor.component.ts

```
import { Component } from '@angular/core';
import {
  ROUTER_DIRECTIVES,
  CanActivate,
  ComponentInstruction,
  OnActivate,
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html'
})
```

Now, let's implement the interface with its required `routerOnActivate` method. As a rule of thumb, all router lifecycle hooks are named after the hook name prefixed by router in lowercase:

app/tasks/task-editor.component.ts

```
export default class TaskEditorComponent implements OnActivate {

  constructor(private title: Title) {}
  routerOnActivate(
    next: ComponentInstruction,
    prev: ComponentInstruction): void {
      this.title.setTitle('Welcome to the Task Form!');
  }
```

```
}
```

Please note how we inject the `Title` type in the class through its constructor and how we later on execute it when the router activates the component once the navigation has finished. Save your changes and reload the application. You will notice how the browser title changes once we successfully access the component after passing the `CanActivate` and `OnActivate` stages.

The CanDeactivate and OnDeactivate hooks

Just like we can filter if the component we are navigating to can be activated, we can apply the same logic when the user is about to leave the current component towards another one located elsewhere in our application. As we saw in case of the CanActivate hook, we must return a Boolean or a Promise resolving to a Boolean in order to allow the navigation to proceed or not. When the CanDeactivate hook returns or is resolved to true, the OnDeactivate hook is executed just like the OnActivate hook after the navigation is accomplished.

In the following example, we will intercept the deactivation stages of the routing lifecycle to first interrogate the user whether he wants to leave the component page or not, and then we will restore the page title if so. For both operations, we will need to implement the CanDeactivate and OnDeactivate interfaces in our component. The code is as follows:

```
export default class TaskEditorComponent implements OnActivate, CanDeactivate,
OnDeactivate {
  constructor(private title: Title) {}

  routerOnActivate(): void {
    this.title.setTitle('Welcome to the Task Form!');
  }

  routerCanDeactivate(): Promise<boolean> | boolean {
    return confirm('Are you sure you want to leave?');
  }
  routerOnDeactivate(): void {
    this.title.setTitle('My Angular 2 Pomodoro Timer');
  }
}
```

Please note that we have removed the (next: ComponentInstruction, prev: ComponentInstruction) arguments from our hook implementations because they were of no use for these examples, but you can access a lot of interesting information through them in your own custom implementations.

Same as the CanActivate hook, the CanDeactivate hook must return a Boolean value or a Promise resolved to a Boolean value in order to allow the routing flow to continue.

The CanReuse and OnReuse hooks

Last but not least, we can reuse the same instance of a component while browsing from one component to another component of the same type. This way, we can skip the process of destroying and instantiating a new component, saving resources on the go.

This requires us to ensure that the information contained in the parameters and stuff is properly handled to refresh the component UI or logic if required in the new incarnation of the same component.

The `CanReuse` hook is responsible for all this, and it tells the `Router` whether the component should be freshly instantiated or whether we should reuse the component in the future calls of the same route. The `CanReuse` interface method should return a Boolean value or a `Promise` resolving to a Boolean value (just like the `CanActivate` or `CanDeactivate` hooks do), which informs the `Router` if it should reuse this component in the next call. If the `CanReuse` implementation throws an error or is rejected from within the `Promise`, the navigation will be cancelled.

On the other hand, if the `CanReuse` interface returns or resolves to true, the `OnReuse` hook will be executed instead of the `OnActivate` hook should the latter exist already in the component. Therefore, use only one of these two whenever you implement this functionality.

Let's see all these in an actual example. When we schedule a task in the task list table and proceed to its timer, we can jump at any time to the generic timer accessible from the top nav bar, thereby loading another timer that is not bound to any task whatsoever. By doing so, we are actually jumping from one instance of the `TimerWidgetComponent` component to another `TimerWidgetComponent` component and the Angular router will destroy and instantiate the same component again. We can save the `Router` from doing so by configuring the component to be reused. Open the `TimerWidgetComponent` module and import the interfaces we will need for this, along with the symbols we were importing already from the `Router` library:

app/timer/timer-widget.component.ts

```
import { Component, OnInit } from '@angular/core';
import { SettingsService, TaskService } from '../shared/shared';
import { RouteParams, CanReuse, OnReuse } from '@angular/router-deprecated';
```

Now, implement the `CanReuse` and `OnReuse` interfaces in the class by adding them to the implements declaration and then proceed to attach the following required interface methods to the class body:

```
routerCanReuse(): boolean {
  return true;
}

routerOnReuse(next: ComponentInstruction): void {
  // No implementation yet
```

```
}
```

Now go to the tasks table, schedule any task, and go to its timer. Click on the **Timer** link in the top nav bar. You will see how the URL changes in the browser but nothing happens. We are reusing the same component as it is. While this saves memory resources, we need a fresh timer when performing this action. So, let's update the `OnReuse` method accordingly, resetting the `taskName` value and the Pomodoro itself:

```
routerOnReuse(): void {
  this.taskName = null;
  this.isPaused = false;
  this.resetPomodoro();
}
```

Reproduce now the same navigation journey and see what happens. Voila! New behavior but same old component.

Advanced tips and tricks

Although we have discussed all that you need to start building complex applications with routing functionalities, there is still a big collection of advanced techniques you can use to take our application to the next level. In the upcoming sections, we will highlight just a few.

Redirecting to other routes

Besides the route definition types we have seen already, there is another `RouteDefinition` type named `Redirect` that is not bound to any named `Route` or `component`, but will rather redirect to another existing `Route`.

So far, we were serving the task list table from the root path, but what if we want to deliver this table from a path named `/tasks` while ensuring that all the links pointing to the root are properly handled? Let's create a redirect route then. We will update the top root router configuration with a new path for the existing home path and a redirect path to it from the new home URL. The code is as follows:

app/app.component.ts

```
...
@RouteConfig([{
  path: '',
  name: 'Home',
  redirectTo: ['TasksComponent']
}, {
  path: 'tasks',
  name: 'TasksComponent',
  component: TasksComponent,
  useAsDefault: true
}, {
  path: 'tasks/editor',
  name: 'TaskEditorComponent',
  component: TaskEditorComponent
}, {
  path: 'timer/...',
  name: 'TimerComponent',
  component: TimerComponent
}])
export default class AppComponent {}
```

The new redirecting route just needs a string path property and a redirectTo property declaring the array of named routes we want to redirect all the requests to.

Note

At the time of this writing, the rotue definitions in the new Router still do not implement support for the `redirectTo` property. Please check the online documentation for a more up-to-date status on the subject.

Tweaking the base path

When we began working on our application routing, we defined the base `href` of our application at `index.html`, so the Angular router is able to locate any resource to load apart from the components themselves. We obviously configured the root `/` path, but what if, for some reason, we need to deploy our application with another base URL path while ensuring the views are still able to locate any required resource regardless of the URL they're being served under? Or perhaps we do not have access to the `HEAD` tag in order to drop a `<base href="/">` tag, because we are just building a redistributable component and do not know where this component will wind up later. Whatever the reason is, we can easily circumvent this issue by overriding the value of the `APP_BASE_HREF` token, which represents the base `href` to be used with our `LocationStrategy` of choice.

Try it for yourself. Open the `main.ts` file where we bootstrap the application, import the required tokens, and override the value of the aforementioned base `href` application variable by a custom value:

app/main.ts

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';
import { provide } from '@angular/core';
import { APP_BASE_HREF } from '@angular/common';

bootstrap(AppComponent, [provide(APP_BASE_HREF, {
  useValue: '/my-apps/pomodoro-app'
})]);
```

Reload the app and see the resulting URL in your browsers.

Finetuning our generated URLs with location strategies

As you have seen, whenever the browser navigates to a path by command of a `routerLink` or as a result of the execution of the navigate method of the `Router` object, the URL showing up in the browser's location bar conforms to the standardized URLs we are used to seeing, but it is in fact a local URL. No call to the server is ever made. The fact that the URL shows off a natural structure is because of the `pushState` method of the HTML5 history API that is executed under the folds and allows the navigation to add and modify the browser history in a transparent fashion.

There are two main providers, both inherited from the `LocationStrategy` type, for representing and parsing state from the browser's URL:

- `PathLocationStrategy`: This is the strategy used by default by the location service, honoring the HTML5 `pushState` mode, yielding clean URLs with no hash-banged fragments (`example.com/foo/bar/baz`).
- `HashLocationStrategy`: This strategy makes use of hash fragments to represent state in the browser URL (`example.com/#foo/bar/baz`).

Regardless of the strategy chosen by default by the `Location` service, you can fallback to the old hashbang-based navigation by picking the `HashLocationStrategy` as the `LocationStrategy` type of choice.

In order to do so, go to `main.ts` and tell the Angular global injector that, from now on, any time the injector requires binding the `LocationStrategy` type for representing or parsing state (which internally picks `PathLocationStrategy`), it should use not the default type, but use `HashLocationStrategy` instead.

It just takes to override a default provider injection:

app/main.ts

```
import 'rxjs/add/operator/map';
import { bootstrap } from '@angular/platform-browser-dynamic';
import AppComponent from './app.component';
import { provide } from '@angular/core';
import {
  LocationStrategy,
  HashLocationStrategy
} from '@angular/common';

bootstrap(AppComponent, [provide(LocationStrategy, {
  useClass: HashLocationStrategy
})]);
```

Save your changes and reload the application, requesting a new route. You'll see the resulting URL in the browser.

Note

Please note that any location-related token in the example is not imported from '@angular/router-deprecated' but from '@angular/common' instead.

Loading components asynchronously with AsyncRoutes

As you have seen in this chapter, each route definition needs to be configured with a `component` property that will inform the router about what to load into the router outlet when the browsers reach that URL. However, we might sometimes find ourselves in a scenario where this component needs to be fetched at runtime or is just the by-product of an asynchronous operation. In these cases, we need to apply a different strategy to pick up the component we need. Here's where a new type of router definition named AsyncRoute comes to the rescue. This specific kind of route exposes the same properties of the already familiar `RouteDefinition` class we have been using along this chapter. It replaces the `component` property with a `loader` property that will be linked to a `Promise` that resolves asynchronously to a component loaded on demand.

Let's see this with an actual example. In order to keep things simple, we will not be importing the component we want to load at runtime, rather we will return it from an asynchronous operation. Open the top root component module and replace the route pointing to `TimerComponent` with this async route definition:

app/app.component.ts

```
...

@RouteConfig([{
    path: '',
    name: 'Home',
    redirectTo: ['TasksComponent']
}, {
    path: 'tasks',
    name: 'TasksComponent',
    component: TasksComponent,
    useAsDefault: true
}, {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
}, {
    path: '/timer/...',
    name: 'TimerComponent',
    loader: () => {
      return new Promise(resolve => {
        setTimeout(() => resolve(TimerComponent), 1000);
      });
    }
  }
])
export default class AppComponent {}
```

The next time we attempt to load any route belonging to the timer branch (either the generic timer accessible from the nav bar or any task-specific timer), we will have to wait until the `Promise` resolves to the component we need. Obviously, the goal of this example is not to

teach how to make things load slower, but to provide a simple example of loading a component asynchronously.

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 definitely shines in the Router module is the vast number of options and scenarios we can cover with such a simple but powerful implementation.

In this chapter, we discussed how to install and provide support for routing in our applications and how to turn any given component into a routing component by decorating it with the router configuration decorator and placing a router outlet in its template, even spreading routers downward in the components tree. We also saw how to define regular routes and some other advanced types such as redirect or async routes. The routing lifecycle has no secrets for us anymore and harnessing its power will open the door to deliver advanced functionalities in our applications with no effort. The possibilities are endless and, most importantly, routing contributes to delivering a better browsing experience.

In the next chapter, we will beef up our task editing component to showcase the mechanisms underlying web forms in Angular 2 and what are the bests strategies to grab user's input with form controls.

Chapter 8. Forms and Authentication Handling in Angular 2

In the previous chapter, we covered routing and this led to security concerns when it came to providing different tiers of content in our application. Enabling user authentication is the first step for introducing relevant features such as publishing forms in our application. However, if we want to build these brand new functionalities, we will need to discover how to set a foundation first. In this chapter, we will see how to build forms and then move on to cover how to leverage those forms to allow users to login and create new content.

As a word of caution about this chapter, we will overview different ways of building forms. All of them are valid, and its use will depend on the goals you're aiming on every moment to fulfill each project requirement.

In this chapter, we will:

- Learn how to create responsive input controls in our forms with directives
- Discuss two-way data binding support in Angular 2
- Bind data models and interface types for forms and input controls
- Design control sets both declaratively and imperatively
- Dive into the alternatives for input validation
- Build our own custom validators
- Develop our own login forms
- Implement general-purpose authorization providers
- Secure areas of our site by requesting user login upfront

Two-way data binding in Angular 2

We mentioned in previous chapters that one of the main differences between Angular 2 and the previous incarnations of the framework is that it does not favor two-way data binding as the core pattern of data management. Well, this is not exactly true. While most of the data management processes in Angular 2 are one way only, form management provides room for two-way data binding by means of the `NgModel` directive.

Let's see all this through an actual example. In the previous chapter, we introduced a new component so we could expand the range of components available in our app in order to have more options for navigating the site, and thus we could better test our router's implementation. This new component, named `TaskEditorComponent`, had no implementation yet and its template featured this layout:

app/tasks/task-editor.component.html

```
<form class="container">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      required>
  </div>

  <div class="form-group">
    <input type="Date"
      class="form-control"
      required>
  </div>

  <div class="form-group">
    <input type="number"
      class="form-control"
      placeholder="Pomodoros required"
      min="1"
      max="4"
      required>
  </div>

  <div class="form-group">
    <input type="checkbox" name="queued">
    <label for="queued"> this task by default?</label>
  </div>

  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TasksComponent']" class="btn btn-danger">Cancel</a>
  </p>
</form>
```

This is a tiny but nifty web form indeed. The component included support for some routing lifecycle hooks in order to serve as a proof-of-concept for the different stages a component goes through in its journey through the navigation pipeline. Apart from that, the form had no life whatsoever—it was just an unanimated creature in the middle of nowhere.

Note

You will see several classnames decorating our forms through the different examples included in this chapter. Unless pointed otherwise, all classnames contained in this chapter are borrowed from the Bootstrap style sheet for styling up our form, for example, `container`, `form-group`, `form-control` and so on. Angular has no relationship with these and they are indeed not required when coding against the framework.

The NgModel directive

Let's infuse some life into it then! One of the good things about implementing two-way data binding support in our form elements is that we do not need to import anything upfront. Angular 2 is smart enough to detect what it needs and the only directive we will need is already supplied out-of-the-box. We are obviously referring to the `NgModel` directive. According to the Angular 2 official documentation:

> *"ngModel binds an existing domain model to a form control. For a two-way binding, use [(ngModel)] to ensure the model updates in both directions."*

In a nutshell, in the very moment we bind an `ngModel` attribute to a form control, the control will watch the value stored at the component `class` property it is bound to and will update itself as soon as the value changes in the model. You might think: this is already done by Angular without any real fanfare. Yes, but the main difference here is that such surveillance is performed in both ways. This means that the class model will update its state as soon as the form control value is updated.

Enough said! It's time for some action. Let's update our task editing component to try this out. Bring up the code of our task editing component and, first of all, please note that it features a `CanActivate` decorator that posed a `passthrough` question to the user. Let's remove it, since we will encounter more secure and elegant ways to provide such functionality later in this chapter. Now, let's add a new member named `taskName`, which will obviously represent a task name!

app/tasks/task-editor.component.ts

```
export default class TaskEditorComponent implements OnActivate, CanDeactivate,
OnDeactivate {
    taskName: string;

    constructor(private title: Title) {}

    // Rest of the component remains unchanged
}
```

Open the associated template and update the first input block to look like this. We will explain all this in a minute:

app/tasks/task-editor.component.html

```
<p>Your task name is {{taskName}}</p>
<div class="form-group">
  <input type="text"
    class="form-control"
    placeholder="Task name"
    [(ngModel)]="taskName">
</div>
```

As you can see, we have attached a `[(ngModel)]` attribute directive into our input control pointing to the string property we just created in the component class, which is also shown on the screen right above the input. Execute the code and change the text field values. You will see how the text entered is updated in real-time on screen.

The syntax of the `ngModel` gives a very good hint to what is it all about. We are blending in a single attribute an event handler and a property binding (hence the combination of brackets plus braces), so we can inject a value into the target control while listening to changes made on the value at the same time. In other words, it is two-way data binding.

Obviously, this is a very simplistic example and we aim to build something more ambitious, so let's leverage this recently gained experience to build something more useful. In the following section, we are going to:

- Link the form to a newly instantiated task object acting as a model
- Populate the model with the values entered in the form
- Persist the changes after validating the data entered
- Redirect the user to the Pomodoros table to see the task just created there

Binding a type to a form with NgModel

Remove the code just added and import the `Task` interface type into our form along with the `TaskService` manager, by appending the following import statement to the top:

app/tasks/task-editor.component.ts

```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';
import {
  Router,
  ROUTER_DIRECTIVES,
  ComponentInstruction,
  CanActivate,
  OnActivate,
  CanDeactivate,
  OnDeactivate } from '@angular/router-deprecated';
import {
  Task,
  TaskService } from '../shared/shared';
```

You might have noticed that we also imported the `Router` type from `angular2/router`. We will need it to redirect the user back to the Pomodoro list page once the new task has been successfully created.

Now, we need to append a new Task-annotated member to our class and declare `TaskService` as a dependency in the constructor, so we can persist the newly created task later. Remove the `taskName` string field we created earlier and update the class with these changes:

app/tasks/task-editor.component.ts

```
export default class TaskEditorComponent implements OnActivate, CanDeactivate,
OnDeactivate {
  task: Task;

  constructor(
    private title: Title,
    private router: Router,
  private taskService: TaskService) {
    this.task = <Task>{};
  }

  // Rest of the component remains unchanged
}
```

We have added a new field to the class representing the `Task` model our form will be bound to. Since `Task` is an interface type, we cannot instantiate it by using the new keyword, since interfaces have no constructor. However, we can take advantage of generics and typecast an empty object to enforce the `Task` interface, as we did in the preceding code.

On the other hand, the types declared in the constructor ensure that the Angular injector will

make them available as class fields for the rest of the component members once it is instantiated.

Ideally, we would just need to link the form data to the object represented by the `task` member of our component class, persist it throughout the application by using the methods already created in the `TaskService` class, and then proceed to the task list right after that. Let's begin by updating our HTML template with the required `ngModel` attributes, including a **Submit** button and a submit handler in the form wrapper tag:

app/tasks/task-editor.component.html

```html
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      [(ngModel)]="task.name">
  </div>

  <div class="form-group">
    <input type="date"
      class="form-control"
      [(ngModel)]="task.deadline">
  </div>

  <div class="form-group">
    <input type="number"
      class="form-control"
      placeholder="Pomodoros required"
      min="1"
      max="4"
      [(ngModel)]="task.pomodorosRequired">
  </div>

  <div class="form-group">
    <input type="checkbox"
      name="queued"
      [(ngModel)]="task.queued">
    <label for="queued"> this task by default?</label>
  </div>

  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TaskList']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

There are two remarkable elements in this piece of code:

- Now each input control features an `ngModel` directive attribute, mapped to one of the properties of the `Task` type represented by the task member of the controller class.

- We have included a **Submit** button in our form, although the form tag does not feature any action attribute, so where are we submitting our form to? The `(submit)` event listener takes care of handling the event by binding an event handler to it.

Our three input fields now benefit from two-way data binding functionality, being each input control pointing to a property exposed by the model object. When submitted, the form will execute the `saveTask()` method located in the body of our component. Let's take a look into this method then. It has not been added already to the class though so please extend our component with a method featuring such a name and append it anywhere in the class right after its constructor:

app/tasks/task-editor.component.ts

```
saveTask() {
  this.task.deadline = new Date(this.task.deadline.toString());
  this.taskService.addTask(this.task);
  this.router.navigate(['TaskList']);
}
```

Note

You have probably raised an eyebrow after watching the first line of code in the `saveTask()` method. Yes, that is weird. We grab the value of the deadline property just to convert it to a string (it was a `Date` object already) and then we turn it into a `Date` object again. There is a reason for this. The `DatePipe` (like the one we use in the `TasksComponent` template) will only take the `Date` objects and these need to be properly formatted since no localization transformation is provided at the time of this writing. The `date` input field does not supply such localization functionality so we need to ensure data consistency across the board by repurposing the data format before saving it. There are better workarounds for this but all of them are basically more verbose and definitely sit outside the scope of our topic here, so we will stick to this quick fix for the rest of the chapter.

Bypassing the CanDeactivate router hook upon submitting forms

Now our component has everything we need in order to reflect the changes made on our model by the form. However, if we attempt to fill out the form with the details of our next task and proceed to save it, we will be confronted with that pesky alert popup we set up in the previous chapter for inviting the users to fill out the form, and that's what we just did now! It's time for a last-minute change then. Let's insert a beacon variable informing whether the form has been updated and successfully saved or not, and use it to skip the popup later where required. The code is as follows:

app/tasks/task-editor.component.ts

```
export default class TaskEditorComponent implements OnActivate, CanDeactivate,
OnDeactivate {
  task: Task;
  changesSaved: boolean;
```

```
  constructor(
    private title: Title,
    private router: Router,
    private taskService: TaskService) {
      this.task = <Task>{};
  }

  saveTask() {
    this.task.deadline = new Date(this.task.deadline.toString());
    this.taskService.addTask(this.task);
    this.changesSaved = true;
    this.router.navigate(['TaskList']);
  }

  routerOnActivate() {
    this.title.setTitle('Welcome to the Task Form!');
  }

  routerCanDeactivate() {
    return this.changesSaved || confirm('Are you sure you want to leave?');
  }

  routerOnDeactivate() {
    this.title.setTitle('My Angular 2 Pomodoro Timer');
  }
}
```

Basically, the `changesSaved` field represents a `boolean` flag that will take a truth value right after persisting the `Task` typed data through the `TaskService` API. This allows the `routerCanDeactivate()` method to either return true as soon as it sees whether the changes have been saved or just throw the confirm popup.

So far so good, but now it's time to get fancy and beautify our form logic a little bit. Validating our input fields is definitely a good starting point and that will lead us to the next stage in our journey through the exciting world of Angular 2 forms.

Tracking control interaction and validating input

Although we are already tracking changes in our input forms through the two-way data binding, we need a better way to watch the overall state of our form. In order to do so, we can take advantage of the `NgForm` directive. The `NgForm` directive keeps track of the state of all input controls found within it. The good news is that such a directive has been present in our example right from the beginning. How? Basically, the `NgForm` directive is configured in its selector to be attached to any `<form>` tag present in our template, providing additional features to our form as real-time tracking of the state of the input fields in respect of validity and user interaction. In other words, if you have a form in your template, you have a `ngForm` directive already.

Tip

You can cancel this automatic binding by appending the `ngNoForm` attribute to any `<form>` tag you do not want to be intervened by Angular.

Let's see all this through a simple example. First, mark all our fields with the HTML5 required attribute:

app/tasks/task-editor.component.html

```
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Task name"
      [(ngModel)]="task.name"
      required>
  </div>

  <div class="form-group">
    <input type="date"
      class="form-control"
      [(ngModel)]="task.deadline"
      required>
  </div>

  <div class="form-group">
    <input type="number"
      class="form-control"
      placeholder="Pomodoros required"
      min="1"
      max="4"
      [(ngModel)]="task.pomodorosRequired"
      required>
  </div>
```

```
  <div class="form-group">
    <input type="checkbox"
      name="queued"
      [(ngModel)]="task.queued">
    <label for="queued"> this task by default?</label>
  </div>

  <p>
    <input type="submit" class="btn btn-success" value="Save">
    <a [routerLink]="['TaskList']" class="btn btn-danger">
      Cancel
    </a>
  </p>
</form>
```

If we attempt to submit the form, automatic alerts will be triggered by the browser applying the validation policies enforced by the HTML5 form validation API. But there is a lot more happening under the hood. If we inspect the code of our form with the browser's developer tools, we will see a myriad of class names decorating the input controls now. Where did all these come from? The answer is simple! The `NgForm` directive, actioning our `<form>` tag, put all these there. Let's inspect the first input field through our browser's dev tools as an example:

```
<input type="text"
  class="form-control ng-untouched ng-pristine ng-invalid"
  placeholder="Task name"
  required="">
```

These classnames (easily identifiable by the `ng-` prefix) give us a very good hint to the different states any given input control can take when wrapped inside the `NgForm` directive. These class bindings are fully reactive to state changes in our input fields. With your dev tools pane open, select any input field, update its value and then empty it again and see how the classnames change, assuming any of the following states:

- **Untouched**: When true, the control has not been interacted with the user
- **Touched**: When true, the control has been interacted with the user
- **Pristine**: The control and its underlying model has not been changed
- **Dirty**: The control and its underlying model has been changed
- **Valid**: The inner model is valid
- **Invalid**: The inner model is not valid

When interacting with our form controls, we will see generated classnames matching these states represented with the `ng-` prefix here and there: `ng-untouched`, `ng-pristine`, `ng-invalid`, and so on.

Tracking changes with local references

Now that we know that our input controls can react to user interactions and model validation, we can take a step further and render more information on screen. For instance, we can style our form in a reactive fashion:

app/tasks/task-editor.component.ts

```
@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html',
  styles: [`
    .ng-valid { border-color: #3c763d; }
    .ng-invalid { border-color: #a94442; }
    .ng-untouched { border-color: #999999; }
  `]
})
...
```

Our form will provide now visual hints of the overall state of each input through its visual layout, but perhaps rendering some messages on screen will compound up the user experience we want to deliver in our app. In order to do so, we need to go into each input control state, and template local references become quite handy for this. They will provide a valuable accessor to the general state handler of our form which is our `ngForm` directive. Let's insert a state message in our form and turn it into a watcher of the state of the first input field, using `ngForm` as a proxy:

app/tasks/task-editor.component.html

```
<form class="container" (submit)="saveTask()">
  <h3>Task Editor:</h3>
  <div class="form-group">
    <p class="text-muted" *ngIf="name.untouched">
      Start here by entering the task name.
    </p>
    <p class="text-success" *ngIf="name.valid && name.touched">
      Well done! That's a good name for a task!
    </p>
    <p class="text-danger" *ngIf="!name.valid && name.touched">
      Oops! You cannot leave the name blank...
    </p>
    <input type="text"
      class="form-control"
      placeholder="Task name"
      [(ngModel)]="task.name"
      #name="ngForm"
      required>
  </div>
...
```

Let's take a closer look at the preceding code. The first block is pretty straightforward: we will render different messages (using the styling provided by Bootstrap, as we did already with the rest of the form) depending of the overall state of the control. In order to refer to the input control, we need to create a local template variable so we can address it from a different element, and so we do by appending the `#name` local template reference in our control. Surprisingly, it is populated with a value though. Local template variables in Angular 2 can be populated with other values, or pointers to other objects and that's exactly what we are doing by pointing `#name` to the `ngForm` string. The `NgForm` directive exports itself under the `ngForm` name, so if we refer to its name from any local template reference, we will gain access to its API and, thus, its state.

Let's wrap up our journey into form building based on classical two-way data-binding. Nevertheless, there is another powerful way to build forms in Angular 2 that has nothing to do with our beloved `[(ngModel)]` directive, although it also provides support for the same functionalities and even extends support for some more features, becoming the preferred way for building forms in Angular 2.

Controls, ControlGroups, and the FormBuilder class

In this chapter, we have seen how to implement two-way data binding in our forms to hook up data entities with input fields. While this approach is perfectly fine, Angular 2 provides a more efficient form model where everything flows in one direction only. There are a lot of upsides for this, but probably the most relevant reason is the remarkable impact on performance that traditional two-way data-binding has in our applications, in comparison to other patterns where information flows in one direction only.

Introducing Controls and Validators

In a nutshell, we might summarize `Controls` in the following statement: A `Control` is the minimum representation of an element being part of a form. Think of a text input or a checkbox as perfect examples of a control. In fact, we have been using Angular 2 controls throughout this chapter every time we tapped into any of the input fields of the Pomodoro task creation form. Angular 2 creates (by default) a control for every form component existing within a `NgForm` directive. Since the form element is attached to this directive, we have been dealing with `Control` objects right from the beginning.

Now, what does it take to create a `Control` object? It is pretty simple really. Once we import the `Control` type from the `angular2/common` barrel, we can create controls in our classes just by instantiating them like this:

```
var firstName = new Control('', Validators.required);
```

The `Control` constructor is pretty simple: the first parameter is the default value our control will assume by default and can allocate any object type. It is usually populated with an empty string when no default value is required. The second parameter expects a function which the `Control` object will use to validate the data input. We can create our own custom validation function, as we will see later in this chapter, or we can take advantage of any of the static functions already created in the `Validators` class, also available at the `angular2/common` barrel. This class exposes the following static validator methods:

- `required`: This requires the control to have an actual non-empty value. It is equivalent to the HTML5 required attribute.
- `minLength(minLength: number)`: The validator will require the control to be populated with a value of a given minimum length.
- `maxLength(maxLength: number)`: The validator will require the control to be populated with a value of a given maximum length.
- `pattern(pattern: string)`: The argument of the pattern validator expects a string containing a regular expression.
- `compose(validators: Function[])`: This is not an actual validator though, but a mixin to combine validators in the array passed as a parameter.
- `composeAsync()`: This is the same as `compose`, but it will expect to convert each function into a promise under the hood and execute all them at once, returning the result of the validation check once all promises have resolved.

Each one of these validators, when interrogated by Angular, will either return a null value indicating that the input is valid or an error map object with further details of the errors causing the invalidation of our control. Don't panic! You will get the full picture later on when we implement some of these validators in a real example.

Controls in the DOM – the ngControl directive

We can create `Controls` in the body of our component class or we can have Angular 2 create and bind them directly into our templates thanks to the `ngControl` attribute directive, formalized in the `NgControlName` class.

Note

Do not confuse the `NgControlName` class with the `NgControl` class, which is the base class that `NgControlName` actually extends from.

This directive can only be used as a child of a `NgForm` directive (which we covered already) or an element decorated with the `NgFormModel` attribute directive, which we will cover shortly. The official Angular documentation describes the `ngControl` using the following statement:

> *"Creates and binds a control with a specified name to a DOM element. This directive can only be used as a child of NgForm or NgFormModel."*

According to this quote, we will usually find the `ngControl` directive decorating an input control like this:

```
<input type="email" ngControl="email">
```

From the moment we introduce a `NgControlName` directive as a named `ngControl` attribute in our input controls, we can access its value, check its validity, and watch it for state changes. The most convenient way is to assign a local template reference pointing to the exported `NgForm` directive and evaluate its properties:

```
<input type="password" ngControl="password" #pwd="ngForm">
<div *ngIf="!pwd.valid">Password is invalid</div>
```

In the preceding example, which is meant to be executed within the context of a form element, we labeled the input control extended already by the `ngControl` directive with a local template reference named `#pwd` pointing to `ngForm`. Angular detects this reference and resets the pointer to the control itself. This way, we can conveniently access and inspect the properties of the `Control` object bound to the `ngControl` directive. You will actually see this pattern quite often when working with forms in Angular 2, as we already did in the previous sections.

Note

It is worth noting that the static methods of the `Validator` class have a counterpart in the form of attribute directives that are sensitive to elements decorated with the `ngControl` directive, so we can apply validation right over them:

```
<input ngControl="taskName" required pattern="[a-zA-Z]*">
<input ngControl="pomodoros" minlength="1" maxlength="5">
```

Grouping controls in the DOM with NgControlGroup

Usually, forms include more than one input control and therefore the data model they represent features several fields or properties. When defining a set of inputs, we can take advantage of the `NgControlGroup` directive. This directive behaves as a reference wrapper for several input controls decorated with the `ngControl` directive, so we can benefit from a single entry point to inspect the values of each different input control contained in the `NgControlGroup`. The following example defines a simple form comprising two input fields wrapped within a `NgControlGroup` whose reference is passed to a component method upon submitting the form:

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

@Component({
  selector: 'hello-form',
  template: `
  <form (submit)="sayHello(fullName)">
    <div ngControlGroup="nameControlsGroup" #fullName="ngForm">
      <div class="form-group">
        <input type="text"
          placeholder="First name"
          ngControl="firstName"
          required>
        <input type="text"
          placeholder="Last name"
          ngControl="lastName"
          required>
      </div>
    </div>
    <input type="submit" value="Say my name">
  </form>
  `
})
export default class SayHello {
  sayHello(controlGroup: NgControlGroup): void {
    if (controlGroup.control) {
      let firstName = controlGroup.control.value.firstName;
      let lastName = controlGroup.control.value.lastName;
      alert(`Hello ${firstName} ${lastName}!`);
    }
  }
}
```

We used local template references to introspect the `NgControlGroup` with itself using `ngForm` as a proxy. The object passed to the `sayHello()` method, which is the reference to the `NgControlGroup` itself and exposes (through the control property) all the information pertaining to the global state of the group: pristine, valid, touched, and so on. We can also inspect the `NgControl` objects contained by the control group by inspecting the `controls` property inside control. The value property of control returns a hash object representation of the actual value of all the input fields contained.

In this example, we are populating the `NgControlGroup` with the `nameControlsGroup` value. This string will become the name of the control group in the context of the wrapping `NgForm` directive. If we slide a local template reference into the `<form>` element and inspect its own `controls` property, we will see an object containing a property with the name given to our group (or groups) pointing to its respective `NgControlGroup` object. Inspecting the value property, on the other hand, will return a hash object with as many properties as control groups found in the form containing each one an object representation of the values of the input controls therein.

Defining control groups imperatively with ControlGroup

In the beginning of this chapter, we saw how to declare controls imperatively in our component controller class. Then, we jumped straight to the DOM and covered how to create controls right in the template and also how to create subsets of controls grouped by the `NgControlGroup` directive.

Grouping controls is a common operation and one that we can do as well from within the component controller class by instantiating `ControlGroup` objects with the help of another type named `FormBuilder`, which creates form objects (in other words, control groups) by parsing the configuration of the input fields of our choice. We will see an actual example in a component class in which, after injecting the `FormBuilder` through the constructor, we access its methods to instantiate a `ControlGroup` with other `ControlGroups` and `Controls` nested:

```
import { Component } from '@angular/core';
import {
  Control,
  ControlGroup,
  FormBuilder,
  Validators } from '@angular/common';

@Component({
  selector: 'my-login',
  providers: [FormBuilder],
  templateUrl: 'my-login.component.html'
})
export class LoginComponent {
  name: Control;
  username: Control;
  password: Control;
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.name = new Control('', Validators.required);
    this.username = new Control('', Validators.required);
    this.password = new Control('', Validators.required);

    this.signupForm = formBuilder.group({
      name: this.name,
      credentials: formBuilder.group({
        username: this.username,
        password: this.password
      })
    });
  }
}
```

In the preceding example, we just created a component featuring a `ControlGroup` representing a typical sign-up form, contained in the `signupForm` field of the `LoginComponent` class. This field is populated in the constructor by the `formBuilder.group()` method with a name control and a credential control that are a control group itself (containing two other controls), with

different flavors of validation. Each one of those controls is an actual instance of the `Control` class. When it comes to instantiating `Control` objects, we can benefit from the `control()` method of the `FormBuilder` class, saving us from importing the `Control` token into the class if we do not need it elsewhere. We can dramatically simplify the implementation of the previous example by refactoring it like this:

```
export class LoginComponent {
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.signupForm = formBuilder.group({
      name: this.formBuilder.control('', Validators.required]),
      credentials: formBuilder.group({
        username: this.formBuilder.control('', Validators.required),
        password: this.formBuilder.control('', Validators.required)
      })
    });
  }
}
```

We can even go a step further and take advantage of the syntax sugar provided by Angular and thus instantiate the controls defined in the `ControlGroup` through a more simplistic syntax:

```
export class LoginComponent {
  signupForm: ControlGroup;

  constructor(formBuilder: FormBuilder) {
    this.signupForm = formBuilder.group({
      name: ['', Validators.required],
      credentials: formBuilder.group({
        username: ['', Validators.required],
        password: ['', Validators.required]
      })
    });
  }
}
```

This is the most simplistic way of instantiating controls, where each control is a property name of a hash object whose value is an array where the first element is the default value we want for our controls followed by the range of validators we want to apply to each control.

These three ways of representing a group of controls as the blueprint for an imaginary sign-up form give us a lot of flexibility to create complex forms from our component controller. What syntax should you choose for your next project? It depends. While this last take on how to instantiate `ControlGroup` and `Control` objects is probably the most popular one because of its simplicity, the former gives us the opportunity to refer to the controls from other ends of our controller class, given the fact that such controls are actually part of its members API. Ultimately, it will depend of where you want your controls to be and from where they are accessible.

Connecting the DOM and the controller with ngFormModel

So we can create forms imperatively from within our component controller. Now what? We need to link all this logic to our HTML template somehow and this is where a new directive comes into play: the `NgFormModel` directive. This directive must be bound to a DOM element (usually the form or any element intervened by `NgForm`) and populated with the name of a `ControlGroup` object binding. From that moment onwards, the `Control` objects attached to the `ControlGroup` object are bonded to the input elements flagged with a `ngControl` directive matching their names. The code is as follows:

```
<form [ngFormModel]="signupForm" (submit)="doSomething($event)">
  <div>
    <input type="text"
      placeholder="Your name"
      ngControl="name">
  </div>

  <div ngControlGroup="credentials">
    <input type="text"
      placeholder="Your username"
      ngControl="username">
    <input type="password"
      placeholder="Your password"
      ngControl="password">
  </div>

  <p>
    <input type="submit" value="Signup">
  </p>
</form>
```

There it is: our beloved `signupForm ControlGroup` is now linked to an actual form whose input controls, handled by `ngControl` directives, are automatically mapped to the `ControlGroup` controls.

Now it is time to translate all this to our Pomodoro project, so let's pull up our sleeves as there's some work to do.

A real example – our login component

Earlier in this chapter, we implemented a basic form to publish our own tasks. In a normal scenario, we will not be willing to leave that functionality open to everyone, so we will want to protect it with a password. Building a login form is the first step and that we will do along this section by developing our next feature: login.

The login feature context

The login functionality can live in parallel to the rest of application functionalities and thus deserves its own folder and facade. The whole implementation will depend on a component named `LoginComponent`, so let's start by creating the basic files we require inside the new login feature folder, located at the same level as shared, tasks, or timer. First, we will create a component with no implementation and its associated template, plus the facade. Do not worry about the implementation details now.

The code is as follows:

app/login/login.component.ts – component controller class

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  ControlGroup,
  Validators,
  Control } from '@angular/common';
import { Router } from '@angular/router-deprecated';

@Component({
  selector: 'pomodoro-login',
  templateUrl: 'app/login/login.component.html'
})
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {}

  authenticate() {}
}
```

app/login/login.component.html – component template

```
<form [ngFormModel]="loginForm"
  class="container"
  (ngSubmit)="authenticate()">

  <h3>This content is password-protected</h3>
  <p>Please enter your credentials below</p>

  <div class="alert alert-danger" *ngIf="notValidCredentials">
    Your credentials are not valid!
  </div>

  <div class="form-group">
    <input type="text"
      class="form-control"
      placeholder="Your username"
```

```
      ngControl="username">
  </div>

  <div class="form-group">
    <input type="password"
      class="form-control"
      placeholder="Your password"
      ngControl="password">
  </div>

  <p>
    <input type="submit"
      class="btn btn-success"
      value="Authenticate"
      [disabled]="!loginForm.valid">
  </p>
</form>
```

app/login/login.ts – feature facade

```
import LoginComponent from './login.component';

export {
  LoginComponent
};
```

Updating the router configuration in our top root component will be a great help in testing the changes we will conduct over the next pages. In order to do so, let's edit different parts of the top root component, as follows:

app/app.component.ts – import statements block

```
import { Component } from '@angular/core';
import { SHARED_PROVIDERS } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import { ROUTER_PROVIDERS, RouteConfig, ROUTER_DIRECTIVES, Router } from
'@angular/router-deprecated';
import { TimerComponent } from './timer/timer';
import { TasksComponent, TaskEditorComponent } from './tasks/tasks';
import { FORM_PROVIDERS } from '@angular/common';
import { LoginComponent } from './login/login';
...
```

app/app.component.ts – RouteConfig params

```
...
@RouteConfig([
  { path: '',
    name: 'Home',
    redirectTo: ['TasksComponent']
  }, {
    path: 'tasks',
    name: 'TasksComponent',
    component: TasksComponent,
    useAsDefault: true
```

```
  }, {
    path: 'tasks/editor',
    name: 'TaskEditorComponent',
    component: TaskEditorComponent
  }, {
    path: 'timer/...',
    name: 'TimerComponent',
    component: TimerComponent
  }, {
    path: 'login',
    name: 'LoginComponent',
    component: LoginComponent
  }
])
export default class AppComponent {}
```

The login form template

It just takes a quick glance at the form to understand the machinery that will be built in the component controller class to bring this little guy to life. The `NgformModel` directive is pointing to a `ControlGroup` named `loginForm` that is yet to be created. The `ControlGroup` will wrap two `Control` objects (username and password), and its state will disable the **Submit** button located at the bottom of the form when not valid. The component will feature a method named `authenticate()` that will handle the form submit event. Last but not least, we have this little chunk of code right before our input controls:

```
<div class="alert alert-danger" *ngIf="notValidCredentials">
   Your credentials are not valid!
</div>
```

As you must have guessed, this message will be displayed in case the credentials entered are not correct. We will perform that validation in the implementation of the `authenticate()` method, as you will see next. There is a new directive in our example: the `ngSubmit` event directive. In Angular's own words, it will signal when the user triggers a form submission. The use of this event directive replaces the `(submit)` event binding we've been using so far.

The login component

Let's conduct the implementation of the login component class now. We currently have the controller class skeleton, but we need to beef up the constructor and the `authenticate()` method:

app/login/login.component.ts

```
...
@Component({
  selector: 'pomodoro-login',
  templateUrl: 'app/login/login.component.html'
})
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {
    this.loginForm = formBuilder.group({
      username: ['', Validators.required],
      password: ['', Validators.required]
    });
  }

  authenticate() {
    let credentials: any = this.loginForm.value;
    this.notValidCredentials = !this.loginForm.valid &&
                                  this.loginForm.dirty;

    if(credentials.username === 'john.doe@mail.com' &&
       credentials.password === 'letmein')
    {
      this.router.navigateByUrl('/');
    } else {
      this.notValidCredentials = true;
    }
  }
}
```

This piece of code conforms pretty much to what we already saw in the previous sections. We have skipped the import statement block in the code snippet for brevity sake, but we can clearly see how we inject the `Router` and `FormBuilder` typed dependencies into our class (also configuring the router object as a private class member) using the constructor injection pattern favored by Angular 2. With an instance of the `FormBuilder` class in place, we can create the `ControlGroup` we require, assign it to the `loginForm` member, and bind it later on to the `NgFormModel` directive decorating the `NgForm` (this is, the form element, remember?) directive awaiting in our template.

Filling out both input controls is required by the grace of the `Validators.required` static

methods found in the `Control` instances, but the piece of code that requires more attention is probably the implementation of the `authenticate()` method. Let's deconstruct it bit by bit:

```
let credentials: any = this.loginForm.value;
this.notValidCredentials = !this.loginForm.valid &&
                              this.loginForm.dirty;
```

Our `authenticate()` method first binds the value of the `loginForm` control group to the credentials variable. Remember that `loginForm.value` will take the shape of a hash object like this:

```
{ username: "", password: "" }
```

On the other hand, the `notValidCredentials` class property evaluates to a `boolean` value as a result of the validity and state of the `loginForm` control group state. The last piece of code is more straightforward and basically entails checking if the credentials are valid in which case the user will be redirected to the index page (feel free to replace the destination path to whatever you feel is more appropriate) or the wrong credentials warning will be displayed on screen.

Note

We are evaluating the credentials against hardcoded values for the sake of simplicity but you should never, and we would like to remark the word *never* in this statement, take this path for checking login information and grant access to sensitive parts of your applications. This is way too insecure and can be easily tampered by malicious hackers with no effort. We will elaborate a little bit on handling authentication in the following pages, but we will stick to hardcoded credentials to minimize complexity in our examples. In a real scenario you should *always* rely on a remote secure authentication API and encrypted tokens to handle session persistence. These concerns are more geared towards backend programming and are obviously beyond the scope of this book.

Applying custom validation to our controls

We have seen already how to apply validators to our controls by adding validator methods when instantiating them, but we still need to give answer to two relevant concerns:

- How to create our custom validators?
- How to combine more than one validator in a single `Control` object?

This section will cover these two issues. A custom validator is basically a function that expects a `Control` object in its signature and will return either a null value (if the `Control` is valid) or a hash object comprised by key/value pairs. We want our login form to check if the username is correct. Since the username is supposed to be an actual e-mail address, maybe it is a good idea to check if the username input contains a valid e-mail address before letting the user move on with submitting the form.

Tip

In a normal scenario, we would be using the pattern validator instead, but the purpose of this example is to showcase the mechanics of a custom validator.

So, let's move on and create an e-mail validation function and append it to the body of our `LoginComponent` class. The code is as follows:

app/login/login.component.ts

```
private emailValidator(control: Control):
{ [key: string]: boolean } {
  if (!/(.+)@(.+){2,}\.(.+){2,}/.test(control.value)) {
    return {
      'emailNotValid': true
    };
  }

  return null;
}
```

The form of the returning object when the control value is not valid is not simple. That object will be appended to the `errors` property of the control annotated with a `Validator` function. This is quite convenient since we can then provide further insights into the source of error when evaluating validity on each input, mostly when the input controls feature more than one validation adapter.

Speaking of the devil, now we need to include two validators in our `Control` so the username `Control` validates its value against the required `Validator` and our new custom `Validator` function. The `Validator` has a static method that will do the trick:

```
this.loginForm = formBuilder.group({
  username: ['',
    Validators.compose([
```

```
      Validators.required,
      this.emailValidator
    ])
  ],
  password: ['', Validators.required]
});
```

Watching state changes in our controls

So far, we saw how we can conduct operations depending on input changes in our controls, but it would be definitely nice to take a more reactive approach depending on certain use-cases. The good news is that both the `ControlGroup` and the `Control` types expose two `EventEmitter` members each, which we can subscribe our own Observers to. Then, we will get prompt notifications every time any `Control` or its wrapping `ControlGroup` object updates either its status (pristine, touched and the like) or value. We are talking about the `statusChanges` and `valueChanges` Observables, which are part of any `Control` or `ControlGroup` object, and operate exposing the same interface we saw when subscribing to HTTP observables in [Chapter 6](#), *Asynchronous Data Services with Angular 2.*

Let's see an actual example. It would be nice to display a real-time visual hint as the user enters his/her username informing if the data entered is an actual username or not. First, update our component controller to include a Boolean field that will be used later on to toggle on and off a visual notification in our template. The code is as follows:

app/login/login.component.ts

```
export default class LoginComponent {
  loginForm: ControlGroup;
  notValidCredentials: boolean = false;
  showUsernameHint: boolean = false;

  constructor(
    formBuilder: FormBuilder,
    private router: Router) {
    this.loginForm = formBuilder.group({
      username: ['', Validators.compose([
        Validators.required,
        this.emailValidator])],
      password: ['', Validators.required]
    });

    const username = this.loginForm.controls['username'];
    username.valueChanges.subscribe(value => {
      this.showUsernameHint = (username.dirty &&
                               value.indexOf('@') < 0);
    });

  }
  // Rest of component class remains the same
  ...
}
```

app/login/login.component.html

```
<div class="form-group">
  <input type="text"
    class="form-control"
    placeholder="Your username"
```

```
      ngControl="username">
    <p *ngIf="showUsernameHint"class="help-block">
      That does not look like a proper username
    </p>
</div>
```

Please notice how we are referring to the username `Control` by traversing the controls property of the `loginForm` object. With a pointer to the username in place, we can subscribe observers to any real-time changes in its value, and therefore act accordingly. In this case, we update the value of the `showUsernameHint` field, if the value entered fulfils the minimum requirements of a username. Whatever value it takes, a visual hint will be displayed or not on screen.

Mocking a client authentication service

Perhaps the word *mocking*, which is pretty common in the context of unit testing, is a bit misleading here but at least serves as a heads-up for what we are going to build now. In the previous section, we implemented a pretty simple user authentication checking but, in a real scenario, we usually delegate all the heavy lifting on an authentication service that wraps all the necessary tools for handling user login, logout, and sometimes authentication for granting access to protected areas of our application.

Next, we will create a simplified version of such service and will put it in charge of handling user login along with the component we just created in the previous section. This service will also manage auth token persistence and provide methods to check if the user has access granted to secure pages.

Before jumping into the code, let's summarize the minimum requirements this service must fulfil:

- We need its API to expose a method to handle user login
- User logout must be handled as well by a public method in this API
- A third method or property should inform if the user is logged in or not so they can proceed to secured pages
- Having an observable property informing of the current state of the active user for authentication will become handy to make the overall UI more reactive

With these specifications in mind, let's build our ideal authentication service. Since this service is component-agnostic and will have an impact on the whole application, we will store it in the services folder of our shared context, applying the naming conventions we already know and exposing it through the shared facade:

app/shared/services/authentication.service.ts

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

@Injectable()
export default class AuthenticationService {

    constructor() {}

    login({username, password}): Promise<boolean> {}

    logout(): Promise<boolean> {}

    static isAuthorized(): boolean {}
}
```

app/shared/shared.ts

```
import Queueable from './interfaces/queueable';
```

```
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  AuthenticationService,
  SettingsService,
  TaskService
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS
};
```

As you can see in the resulting facade, the new service will become part of the
`SHARED_PROVIDERS` group token. Then, it will be available for our application injector, since
this symbol is being declared in the providers array of our root component.

Back to the service class, we imported the Injectable decorator. As you know, we will need it
if we want our `AuthService` class to be automatically instantiated and injected as a singleton in
our components by Angular (in case our class requires its own dependencies in the future).
We also import the `EventEmitter` class, which we will cover later in this section.

In the body of the `AuthenticationService` class, we have defined an empty constructor and
three methods with no implementation (one of them being static). While the names give a very
good hint of the purpose of each method, perhaps the last one requires some more
elaboration: The `isAuthorized()` method will inform if the user has permissions to access
secured pages. The reason why it is static is because we will need to use it in some areas
where Angular's dependency injection machinery cannot reach so no automatic provider
injection is available.

Our first requirement was to provide a public method to handle user login. Let's go for it. Get

back to the `AuthenticationService` module and extend the `login` method with the following implementation:

app/shared/services/authentication.service.ts

```
login({username, password}): Promise<boolean> {
  return new Promise(resolve => {
    let validCredentials: boolean = false;

    // @NOTE: In a real scenario this check
    // should be performed against a web service:
    if (username === 'john.doe@mail.com' &&
        password === 'letmein') {
          validCredentials = true;
          window.sessionStorage.setItem('token', 'eyJhbGciOi');
    }

    resolve(validCredentials);
  });
}
```

As you can see from the comments inline in the code, we are not submitting data for validation to a remote web service although we should definitely do. Please recall the warning we raised in previous chapters: you should never implement user validation this way. Having said that, let's review this implementation. In the first place, there is something that draws our attention: the returning type. This method is supposed to return a `Promise` and there is a good reason for that. Usually, you would also want to implement an async HTTP connection to a remote service so you can send the user credentials and wait for a response. Hence, we use the asynchronous interface in the form of a returning `Promise`, which resolves to a Boolean value informing if the credentials provided are good to access the system or not. On the other hand, the method signature is not an annotated argument, but a deconstructed object informing that this method will expect any type or object in its payload containing both username and password properties. Last but not least, right after conducting our fake user validation, we store a random token onto the user's browser using the browser's own session storage layer. This is a common way of handling authentication and user session persistence nowadays, with the sole difference that the token is usually sent in the body of the server response and thereafter is sent back in the request headers on every information request made to the server.

Conducting a server-side implementation is beyond the scope of this book, so we will not explore that topic in greater depth. You can refer to the Packt library for further reference.

Now that we know how to handle user login, implementing a user `logout()` method that literally reverses what the previous `login()` method did is pretty easy:

app/shared/services/authentication.service.ts

```
logout(): Promise<boolean> {
  return new Promise(resolve => {
    window.sessionStorage.removeItem('token');
```

```
    resolve(true);
  });
}
```

Our third requirement was to provide a property or method that would tell us if the user is authenticated, so they can proceed to secured pages. Keeping in mind that user permission is tied to the existence or absence of a security token stored in the browser storage layer, the method logic is really simple:

```
static isAuthorized(): boolean {
  return !!window.sessionStorage.getItem('token');
}
```

We will discuss this method and the rationale behind its static annotation later on. Now, let's move into the last bit of our service: providing an Observable that allows UI elements and other application clients to subscribe to updates in the user status. First, we will create a public EventEmitter member, which we can use to send notifications every time the user logs in and out so that other classes and components can subscribe to it as mere observers and react to those events. Obviously, the login and logout methods will be updated to also send the corresponding notifications to the observers depending on the actions taken and the user state at all times.

With all these changes, this is the final layout of our injectable authentication service:

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

@Injectable()
export default class AuthenticationService {
  userIsloggedIn: EventEmitter<boolean>;

  constructor() {
    this.userIsloggedIn = new EventEmitter();
  }

  login({ username, password }): Promise<boolean> {
    return new Promise(resolve => {
      let validCredentials: boolean = false;

      // @NOTE: In a normal scenario this check
      // should be performed against a web service:
      if (username === 'john.doe@mail.com' &&
        password === 'letmein') {
        validCredentials = true;
        window.sessionStorage.setItem('token', 'eyJhbGciOi');
      }

      this.userIsloggedIn.emit(validCredentials);
      resolve(validCredentials);
    });
  }

  logout(): Promise<boolean> {
    return new Promise(resolve => {
```

```
      window.sessionStorage.removeItem('token');
      this.userIsloggedIn.emit(false);
      resolve(true);
    });
  }

  static isAuthorized(): boolean {
    return !!window.sessionStorage.getItem('token');
  }
}
```

Exposing our new service to other components

With the authentication service provider now available from our application injector, we can begin hooking it up in other components: the login feature is the most logical starting point. First, open the `LoginComponent` code unit and import the new `AuthenticationService` token so that we can properly use its type to inject it into the component:

app/login/login.component.ts

```
import { Component } from '@angular/core';
import {
  FormBuilder,
  ControlGroup,
  Validators,
  Control } from '@angular/common';
import { Router } from '@angular/router-deprecated';
import { AuthenticationService } from '../shared/shared';
...
```

In the same code unit, let's now update the constructor payload with a new argument annotated with the `AuthenticationService` token, so the Angular 2 DI machinery becomes aware that this module requires that type to be injected:

```
constructor(
  formBuilder: FormBuilder,
  private router: Router,
  private authService: AuthService) {
  // Rest of constructor implementation remains unchanged
    ...
}
```

With all the code in place, now we can replace our `authenticate()` method to remove the business logic:

```
authenticate() {
  let credentials: any = this.loginForm.value;
  this.notValidCredentials = !this.loginForm.valid &&
                              this.loginForm.dirty;

  this.authenticationService.login(credentials).then(success => {
    if (success) {
      this.router.navigateByUrl('/');
    } else {
      this.notValidCredentials = true;
    }
  });
}
```

Blocking unauthorized access

With all this in place, it's time to actually prevent unlogged users from accessing protected content. In our case, it just entails protecting the task editing form component from unauthorized requests. In the previous chapter, we saw how to allow or prevent component instantiation by means of `Router` hooks.

With that knowledge to hand, protecting the task editor component from undesired visits becomes quite simple. Open the `TaskEditorComponent` file, import our new `AuthenticationService` provider, and check whether the user is authorized by binding the execution of the static `isAuthorized()` method to the `CanActivate` decorator:

app/tasks/task-editor.component.ts

```
...
// Other import statements remain as they are already
import {
  Task,
  TaskService,
  AuthenticationService } from '../shared/shared';

@Component({
  selector: 'pomodoro-tasks-editor',
  directives: [ROUTER_DIRECTIVES],
  providers: [Title],
  templateUrl: 'app/tasks/task-editor.component.html',
  styles: [`
        .ng-valid { border-color: #3c763d; }
        .ng-invalid { border-color: #a94442; }
        .ng-untouched { border-color: #999999; }
     `]
})
@CanActivate(AuthService.isAuthorized)
export default class TaskEditorComponent implements OnActivate, CanDeactivate,
OnDeactivate {
  // The class implementation remains the same
  ...
}
```

And that's it! Now, any unlogged user attempting to access the protected task editor component will get nothing! If you look carefully at the `@CanActivate()` decorator, you will understand why we defined the `isAuthorized()` method of the `AuthenticationService` as static. The reason relies on the fact that we can only inject dependency singletons in our components but not in decorators. While this is not exactly true (we can leverage the `provide()` injector to bring the desired singleton although the code required is nowhere near as simple or neat), the truth is that this implementation is simple: providing the same level of effectiveness in a neat and clear fashion.

Note

The ideal scenario would be to inject both the `AuthenticationService` and the `Router` providers in the `CanActivate` implementation, and then redirect the user to the login page should the user is not logged in.

Unfortunately, at the time of wrapping up the writing of this book, there is still no formal support for dependency injection in the context of the `CanActivate` router hook. However, this issue is part of the features that will become part of Angular 2 Final. It is quite likely that the `@CanActivate` decorator will be replaced by an analogue instance method of a parent routing component once Angular 2 becomes final eventually. Please refer to the official documentation.

Making the UI reactive to the user authentication status

All right, so unauthorized users cannot access the task editor form component. However, having an unresponsive link in our main toolbar is definitely not good, so we should leverage the Observable features of the `AuthenticationService` to flip the UI whenever there is a change in the user login status.

Right now, the nav bar features the `Login` link that leads the user login form page. What we want to do is to hide the `Publish Task` link and make sure we only display it when the user is logged in, no matter where and how this login procedure was undertaken. On the other hand, we also want to offer the end user a `Logout` link when logged in, so they can shut down his session in confidence. This logout link should be made available for logged in users only. Access to the protected component by hardcoding URLs is not a concern, since the `@CanActivate` decorator will do its job to keep undesired users away.

Now that we have described the requirements, let's put them into practice. Open the top root component file and update its implementation (it remained empty until now). We will need the `AuthenticationService` and the `Router` dependencies to do so, so make sure to import them at the top of the file:

app/app.component.ts

```
import { Component } from '@angular/core';
import {
  SHARED_PROVIDERS,
  AuthenticationService } from './shared/shared';
import { HTTP_PROVIDERS } from '@angular/http';
import {
  ROUTER_PROVIDERS,
  RouteConfig,
  ROUTER_DIRECTIVES,
  Router } from '@angular/router-deprecated';
// Rest of import statements remain the same
…
```

With the tokens properly declared in the import statements, we can move on and provide an implementation for the `AppComponent` class:

app/app.component.ts

```
...
export default class AppComponent {
  userIsLoggedIn: boolean;

  constructor(
    private authenticationService: AuthenticationService,
    private router: Router) {
    authenticationService.userIsloggedIn.subscribe(isLoggedIn => {
      this.userIsLoggedIn = isLoggedIn;
    });
```

```
    }

    logout($event): void {
      $event.preventDefault();

      this.authenticationService.logout().then(success => {
        if (success) {
          this.router.navigateByUrl('/');
        }
      });
    }
}
```

We have declared a `userIsLoggedIn` Boolean field, which will change its value every time the observable `userIsloggedIn` member of the injected `AuthenticationService` type changes its value. We also injected the `Router` type and created a new component method named `logout()` that will wipe out the user session and redirect the user to the root page upon signing out from the application.

This gives us the chance to wrap up the application UI by updating the root component template to make the sensible links fully reactive to these changes:

app/app.component.html

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro App</strong>
    </div>
    <ul class="nav navbar-nav navbar-right">
      <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
      <li><a [routerLink]="['TimerComponent']">Timer</a></li>
      <li *ngIf="userIsLoggedIn">
        <a [routerLink]="['TaskEditorComponent']">Publish Task</a>
      </li>
      <li *ngIf="!userIsLoggedIn"><a [routerLink]="
['LoginComponent']">Login</a>
      </li>
      <li *ngIf="userIsLoggedIn">
        <a href="#" (click)="logout($event)">Logout</a>
      </li>
    </ul>
  </div>
</nav>
<router-outlet></router-outlet>
```

Give it a try! Reload the application, check the links available at the nav bar, head over to the login page, proceed to login with the credentials, and check the nav bar again... Magic!

Running the extra mile on access management

Apparently, we have everything that it takes to move on with our application. However, as our application grows and more areas need to be protected, we will find ourselves facing the burden of toggling visibility on more and more links and having to enable access and activation of components one by one by implementing the `@CanActivate` decorator on each.

Obviously, this scales up just badly, so it would be great to rely on a one-size-fits-all solution instead. Unfortunately, at the time of writing, the Angular 2 framework still does not provide a feasible solution to tackle with this concern. On the other hand, and according to modern UX, most of the time the expected behavior is to provide the user with as many browsing alternatives as possible and redirect the user to the login page only where required.

In this last section, we will introduce a generic workaround for this, based on the following criterion: protecting areas of content as a whole by wrapping them inside child routes that will redirect the user to the login page whenever unauthorized access is detected, where parameters such as the location of the login path are fully configurable from our solution.

Building our own secure RouterOutlet directive

Our workaround is based on developing our very own `RouterOutlet` directive by extending the `RouterOutlet` class baked in Angular 2, which will be slightly rewritten to override the base directive's default behavior when it comes to proceeding to activate (or not) the requested component. All of this based on the current user login status.

To do so, we will create a new directive in our shared context, which will be used across the application. So we need to expose it in the shared facade as well.

app/shared/directives/router-outlet.directive.ts

```
import {
  Directive,
  ViewContainerRef,
  DynamicComponentLoader,
  Attribute,
  Input } from '@angular/core';
import {
  Router,
  RouterOutlet,
  ComponentInstruction } from '@angular/router-deprecated';
import { AuthenticationService } from '../shared';

@Directive({
  selector: 'pomodoro-router-outlet'
})
export default class RouterOutletDirective extends RouterOutlet {
  parentRouter: Router;
  @Input() protectedPath: string;
  @Input() loginUrl: string;

  constructor(
    _viewContainerRef: ViewContainerRef,
    _loader: DynamicComponentLoader,
    _parentRouter: Router,
    @Attribute('name') nameAttr: string) {

    super(_viewContainerRef, _loader, _parentRouter, nameAttr);
    this.parentRouter = _parentRouter;
  }

  activate(nextInstruction: ComponentInstruction): Promise<any> {
    let requiresAuthentication =
        this.protectedPath === nextInstruction.urlPath;

    if (requiresAuthentication &&
        !AuthenticationService.isAuthorized()) {
      this.parentRouter.navigateByUrl(this.loginUrl);
    }

    return super.activate(nextInstruction);
  }
```

```
}
```

Here, we are creating a new directive that extends from the `RouterOutlet` directive we have been using all this time in our application. This directive needs to be made available for use from the other feature contexts of our application:

app/shared/shared.ts

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

import RouterOutletDirective from './directives/router-outlet.directive';

const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  AuthenticationService,
  SettingsService,
  TaskService
];

const SHARED_DIRECTIVES: any[] = [
  RouterOutletDirective
];

export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS,

  RouterOutletDirective,
  SHARED_DIRECTIVES
};
```

Back to the directive file, we can see it inherits the same constructor of the inherited `RouterOutlet` constructor. Thus, we will import the same required tokens so that we can

properly declare the constructor dependencies and call the superclass constructor with `super()`. The code is as follows:

app/shared/directives/router-outlet.directive.ts

```
constructor(
  _elementRef: ElementRef,
  _loader: DynamicComponentLoader,
  _parentRouter: Router,
  @Attribute('name') nameAttr: string) {

  super(_elementRef, _loader, _parentRouter, nameAttr);
  this.parentRouter = _parentRouter;
}
```

In the constructor body, we do not only inject the dependencies, we also assign the `Router` instance in a class member for future use. The core of the solution relies on overriding the implementation of the native `activate()` method, where we basically introduce an authentication check (that is why we also import the `AuthenticationService` at the top of the script) to see if the recently required component lives in the domain of the protected path. In that case, we will redirect the user to the login page location should the authentication token is not available thanks to the static method `isAuthorized()` exposed by the `AuthenticationService`. In any event, the method will finally return the result of the superclass' activate method, represented by a Promise object. The code is as follows:

```
activate(nextInstruction: ComponentInstruction): Promise<any> {
  let requiresAuthentication =this.protectedPath === nextInstruction.urlPath;

  if (requiresAuthentication &&
      !AuthenticationService.isAuthorized()) {
    this.parentRouter.navigateByUrl(this.loginUrl);
  }

  return super.activate(nextInstruction);
}
```

Where do we fetch these `protectedPath` and `loginUrl` parameters? As you saw already at the beginning of this section, we are exposing two input parameters on this directive, which will make its instances look like this:

```
<pomodoro-router-outlet protectedPath="edit" loginUrl="/login">
</pomodoro-router-outlet>
```

So, open up the top router component file and replace the current `RouterOutlet` directive instance in the template with a few lines of code, right after importing the `SHARED_DIRECTIVES` symbol in the directives property of the component decorator:

app/app.component.ts

```
import { Component } from 'angular2/core';
import {
```

```
  SHARED_PROVIDERS,
  AuthenticationService,
  SHARED_DIRECTIVES } from './shared/shared';
...

@Component({
  selector: 'pomodoro-app',
  directives: [ROUTER_DIRECTIVES, SHARED_DIRECTIVES],
  ...
})
```

app/app.component.html

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">
      <strong class="navbar-brand">My Pomodoro App</strong>
    </div>
    <ul class="nav navbar-nav navbar-right">
      <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
      <li><a [routerLink]="['TimerComponent']">Timer</a></li>
      <li *ngIf="userIsLoggedIn">
        <a [routerLink]="['TaskEditorComponent']">
          Publish Task
        </a>
      </li>
      <li *ngIf="!userIsLoggedIn">
        <a [routerLink]="['LoginComponent']">Login</a>
      </li>
      <li *ngIf="userIsLoggedIn">
        <a href="#" (click)="logout($event)">Logout</a>
      </li>
    </ul>
  </div>
</nav>
<pomodoro-router-outlet
  protectedPath="tasks/editor"
  loginUrl="login">
</pomodoro-router-outlet>
```

Last but not least, in order to try out this solution, we still need to do two things: *remove* (or comment out) the `@CanActivate()` decorator in the task editor component and then tweak the `routerCanDeactivate()` router hook to look like this:

app/tasks/task-editor.component.ts

```
routerCanDeactivate(
  next: ComponentInstruction,
  prev: ComponentInstruction) {
  return !AuthenticationService.isAuthorized() ||
    this.changesSaved ||
    confirm('Are you sure you want to leave?');
}
```

This way, we can smoothly deactivate the component if the user is not logged in. Why should

we do this? Basically, this is the flow the user will follow:

- The user asks for a protected component
- The router directive checks if the user is authenticated
- If logged in already, the directive does nothing and the component is activated
- If not, although the component will be activated, a redirection will be performed at the same time and the user will land safely on the comfort of the login page

In order to properly try out this solution, remove the conditional from the route setting. We want to actually display the link but redirect the user if not logged in. The code is as follows:

app/app.component.html

```
...
<ul class="nav navbar-nav navbar-right">
  <li><a [routerLink]="['TasksComponent']">Tasks</a></li>
  <li><a [routerLink]="['TimerComponent']">Timer</a></li>
  <li>
    <a [routerLink]="['TaskEditorComponent']">
      Publish Task
    </a>
  </li>
  <li *ngIf="!userIsLoggedIn">
    <a [routerLink]="['LoginComponent']">Login</a>
  </li>
  <li *ngIf="userIsLoggedIn">
    <a href="#" (click)="logout($event)">Logout</a>
  </li>
</ul>
...
```

Note

There is an important caveat in this solution: the protected component will be actually rendered on screen before bouncing the user to the login page (whenever login is required). This is definitely not good, since there is a chance that the protected contents will flicker on screen if the secure component instantiation takes longer than expected or either the `canDeactivate()` method poses some conditions in order to move on (hence the change we introduced.

Otherwise, the transition will be so fast that the chances are that the end user will not even notice these components have been instantiated. In any event, use this with care and watch out for the `CanDeactivate` function in all your implementations.

Summary

This was quite a long and dense complicated chapter, without any doubt. The most important takeaway from this chapter was that there is not one but many alternatives when it comes to designing and implementing forms in Angular 2. Some of them are more declarative and some others are mode imperative. When should you use one or favor another? As we said already, it depends on where and how you want to access and tackle the issue of accessing the `Control` and `ControlGroup` state and validity properties.

User authentication was also covered in this chapter and we introduced different alternatives for catering with protected areas of our site. Stay tuned and keep an eye on the latest accomplishments made in the Angular 2 arena, since it is quite likely that new workarounds for the most common use cases will spring out down the track.

Now, get ready to confront the last leg of our journey into Angular 2, where we will provide some coverage on the framework support for animations before wrapping up everything by learning some unit testing techniques—all in the last chapter of the book.

Chapter 9. Animating Components with Angular 2

Nowadays, animations are one of the cornerstones of modern user experience design. Far from just representing a visual eye candy for beautifying the UI, they have become an important part of the visual narrative. Animations pave the road to convey messages in a non-intrusive way, becoming a cheap but powerful tool for informing the user about the underlying processes and events that happen while we interact with our application. The moment an animation pattern becomes widespread and the audience embraces it as a modern standard, we gain access to a priceless tool for enhancing our application's user experience. Animations are language-agnostic, are not necessarily bound to a single device or environment (web, desktop or mobile) and are pleasant to the eye of the beholder when used wisely. In other words, animations are here to stay and Angular 2 has a strong commitment to this aspect of modern visual development.

With all modern browsers embracing the newer features of CSS3 for animation handling, Angular 2 offers support for implementing imperative animation scripting through an incredibly easy but powerful API. This chapter will cover several approaches to implementing animation effects, moving from leveraging plain vanilla CSS for applying class-based animations, to implementing script routines where Angular 2 takes full responsibility for handling DOM transitions.

In this chapter we will:

- Create animations with plain vanilla CSS
- Leverage class-named animation with the `ngClass` directive to better handle transitions
- Look at Angular's built-in CSS hooks for defining styles for each transition state
- Animate components with the `CssAnimationBuilder` API
- Design directives that handle animation
- Introduce ngAnimate 2.0

Note

As a word of caution, bear in mind that the current implementation of animation in Angular 2 at the time of writing is, to a certain extent, temporary. In that sense, all the functionalities described in this chapter could probably be deprecated in the long run as the Angular 2 codebase matures. However, for now, there is no reason to hold ourselves back and leverage the animation modules already available in the framework. This will give us the power and functionality required to enhance the overall user experience of our applications.

Creating animations with plain vanilla CSS

The inception of CSS-based animation set an important milestone in modern web design. Before that, we used to rely on JavaScript to accommodate animations in our web applications by manipulating DOM elements through complex and cumbersome scripts based on intervals, timeouts, and loops of all sorts. Unfortunately, this was neither maintainable nor scalable.

Then modern browsers embraced the functionalities brought by the recent CSS `transform`, `transition`, `keyframes`, and `animation` properties. This became a game changer in the context of web interaction design in recent times. While support for these techniques in browsers such as Microsoft Internet Explorer is far from optimal, the rest of the browsers in store (including Microsoft's very own Edge) provide full support for these CSS APIs.

Note

MSIE provides support for these animation techniques only as of Version 10.

We assume that you have a broad understanding of how CSS animation works in regards of building keyframe-driven or transition-based animations, since providing coverage to these techniques is obviously out of the scope of this book. As a recap, we can highlight the fact that CSS-based animation is usually implemented by any of these approaches, or even a combination of both:

- **Transition properties**, that will act as Observers of either all or just a subset of the CSS properties applied to the DOM elements impacted by the selector. Whenever any of these CSS properties is changed, the DOM element will not take the new value right away, but will experience a steady transition into its new state.
- **Named keyframe**, animations, where we define different steps of the evolution of one or several CSS properties under a unique name, which will populate later on an `animation` property of a given selector, being one able to set additional parameters as the delay, duration of the animation tweening or the number of iterations that such animation is meant to feature.

As we can see in the two aforementioned scenarios, the use of a CSS selector populated with animation settings is the starting point for all things related to animation, and that is what we will do now: let's build a fancy pulse animation to emulate a heartbeat-style effect in the bitmap that decorates our Pomodoro timer.

We will use a keyframe-based animation this time, so we will begin by building the actual CSS routine in a separate style sheet. The entire animation is based on a simple interpolation where we take an object, scale it up by 10 percent and scale it back down again to its initial state. This keyframe-based tweening is then named and wrapped in a CSS class named `pulse`, which will execute such animation in an infinite loop where each iteration takes 1 second to complete.

All the CSS rules for implementing this animation will live in an external style sheet part of the timer widget component, within the timer feature folder:

app/timer/timer-widget.component.css

```css
@keyframes pulse {
  0% {
    transform: scale3d(1, 1, 1);
  }

  50% {
    transform: scale3d(1.1, 1.1, 1.1);
  }

  100% {
    transform: scale3d(1, 1, 1);
  }
}

.pulse {
    animation: pulse 1s infinite;
}
```

As for this point on, any DOM element (in the `TimerWidgetComponent` template) annotated with this class name will visually beat like a heart. This visual effect is actually a good hint that the element is undertaking some kind of action, so applying it to the main pomodoro icon bitmap in our pomodoro timer widget when the countdown is on will help convey the feeling that an activity is currently taking place in a lively fashion.

Thankfully, we have a good way to apply such effect only when the countdown is active. We use the `isPaused` binding in the `TimerWidgetComponent` template. Binding its value to the `NgClass` directive in order to render the `classname` only when the component is not paused will do the trick, so just open the timer widget code unit file and add a reference to the style sheet we just created and apply the directive as described previously:

app/timer/timer-widget.component.ts

```ts
...
@Component({
  selector: 'pomodoro-timer-widget',
  styleUrls: ['app/timer/timer-widget.component.css'],
  template: `
    <div class="text-center">
      <img src="/app/shared/assets/img/pomodoro.png"
        [ngClass]="{ pulse: !isPaused }">
      <h3><small>{{ taskName }}</small></h3>
      <h1> {{ minutes }}:{{ seconds  | number: '2.0' }} </h1>
      <p>
        <button (click)="togglePause()" class="btn btn-danger">
        {{ buttonLabelKey | i18nSelect: buttonLabelsMap }}
        </button>
      </p>
```

```
        </div>`
})
...
```

And that's it! Run our pomodoro app and click on the Timer link at the top to reach the timer component page and check the visual effect live after starting the countdown. Stop it and resume it again to see the effect applied only when the countdown is active.

Handling animation with CSS class hooks

As we have just seen in the previous section, applying visual effects based on CSS classes is a breeze thanks mostly to the flexibility we have for adding custom class names in Angular 2.

On top of that, Angular provides support for animation class hooks, a functionality that was already available in Angular 1.x, under a different incarnation though. Basically, the mechanics is as follows: DOM elements managed by template-driven directives (`NgSwitch`, `NgFor`, or `NgIf`) can be decorated with the `ng-animate` classname. From that very moment, such elements will feature additional class names depending on the stage of the animations applied to that DOM element in the context of the wrapping component lifecycle.

This last statement might sound a bit odd and daunting, so let's see all this in action through an actual example. Open our `TasksComponent` template and update the row tag of the tasks list by adding a class named `ng-animate` to the markup. Then, do the same with the `queued` label displayed when the task model is lined up in our tasks queue. The code is as follows:

app/tasks/tasks.component.html

```html
<tr *ngFor="let task of tasks; let i = index" class="ng-animate">
  <th scope="row">{{i}}
    <span *ngIf="task.queued" class="label label-info ng-animate">
      Queued
    </span>
  </th>
  <!-- the rest of the template remains untouched -->

</tr>
```

Save everything and then run again the application while inspecting the code in the browser dev tools. Well, apparently nothing happens. But we are talking about animations here, and as a matter of fact the underlying process is expecting exactly that, so let's decorate the `ng-animate` class with some animation-related styling defined in the component's associated style sheet:

app/tasks/tasks.component.css

```css
h3, p {
  text-align: center;
}
.table {
  margin: auto;
  max-width: 860px;
}
.ng-animate {
  transition: all 0.3s ease-in;
}
```

The CSS rule defined previously will force any styling change applied to the DOM element to

take place in 10 seconds following an ease-in algorithm curve when applied. Run the code again and inspect the code: apparently we're now into something: all elements flagged with the `ng-animate` class now feature some classes that temporarily decorate the element. These classes are `ng-enter` and `ng-enter-active`, where the first one is enabled by default when rendering the element and the latter applied straight away. After 10 seconds, which is the scope of the transition we defined in the CSS rule, both class names will disappear, leaving the `ng-animate` class as the only track of its now extinct existence.

Basically, we have the same behavior we implemented by hand in the previous section, with the sole exception that the class name's binding is operated this time by Angular 2 itself. With all this in mind, let's repurpose the style sheet a little bit to leverage the brief existence of these `classnames` in our DOM and the transition property to make a nice fade-in effect occur upon loading. Replace the 10 seconds transition period by a shorter 0.3 second value (300 milliseconds) and let's define opacity values for the initial `ng-enter` class name and the final `ng-enter-active` class names. The code is as follows:

app/tasks/tasks.component.css

```
...
.ng-animate {
  transition: all 0.3s ease-in;
}
.ng-enter {
  opacity: 0;
}
.ng-enter-active {
  opacity: 1;
}
```

Save and rerun the code examples. Now, you will see how the task list smoothly fades in on screen. The same applies to the blue label informing if any given task has been queued up or not.

Class hooks available

The class names we have just seen are known in Angular-land as `class hooks`, which resonates from the directive or routing lifecycle events we already covered, and each one of them will only exist while the animation takes place. We have four class hooks:

- `ng-enter`: This will be applied by Angular's DOM renderer to any element flagged with the `ng-animate` class name upon being attached to the view. It usually wraps the styling we require our component to feature by default.
- `ng-enter-active`: This classname is temporarily attached to the element on runtime right before starting the CSS animation and is removed automatically, along with the `ng-enter` class name, when the animation is completed. It usually defines the styling we strive our component to assume by the end of the animation which was previously reset by `ng-enter`.
- `ng-leave`: Think of this class hook as the counterpart of `ng-enter`, but it takes place

when the element is about to be detached from the view.

- `ng-leave-active`: This is same as `ng-enter-active`, but it takes place when the element is about to be detached from the view. The `classname` is applied to the element and will be removed, along with ng-leave, once the `transition` is completed and before the element is removed from the DOM.

Tip

Do not ever forget that this technique is only available for DOM elements that are handled by the `DomRenderer` type, which is a low-level class in charge of creating, updating, or removing nodes and views among other tasks. In our case, elements decorated with the `NgIf, NgFor, or NgSwitch` directives are the only feasible candidates for it.

Animating components with the AnimationBuilder

If you ever decide to investigate how the CSS class hooks triggered by the `ng-animate` class binding work under the hood, you will be positively surprised by the fact that all the heavy lifting is carried out by an instance of the `CssAnimationBuilder` class instantiated through the `AnimationBuilder` API.

The `AnimationBuilder` class (which is an injectable type and therefore subject to be imported through the constructor of our components) is a factory type whose API provides access to instantiate more specialized animation builders such as the `CssAnimationBuilder` class. This type has a very broad and powerful API whose methods allow us to add or remove CSS class names in order to trigger transitions or animations, or even configure by hand animation parameters such as styles, duration, or delay on our DOM elements of choice. In this sense, we can define general purpose animation handlers and then use them as animation adapters for any DOM element. In that sense, animation handlers created by the `CssAnimationBuilder` are agnostic of the DOM elements. Therefore, a single animation adapter can be applied to one or many HTML elements.

So, in order to create our own animations programmatically using only JavaScript, we just need an instance of the `AnimationBuilder` type, which we will use to instantiate a `CssAnimationBuilder` for creating a specific (HTML node-agnostic) animated transition. Last but not least, an accessor type to the DOM elements we want to animate. Sounds daunting? A quick and easy example will clarify all this.

A good way to put all these to the test is to introduce a new visual effect on our Pomodoro timer: a rendering animation effect when the component is loaded, so every time we activate the `PomodoroTimer` route, the component gets loaded gracefully by fading in on screen.

Let's begin by importing new tokens in the block of import statements of `TimerWidgetComponent`. We will need to fetch `ElementRef` from angular/core, since it will give us access to the DOM element we want to animate. Then we will have to bring the `AnimationBuilder` symbol, which will be used to instantiate a `CssAnimationBuilder` object. The latter is also imported so we can properly annotate our class members. The code is as follows:

app/timer/timer-widget.component.ts

```
import { Component, OnInit, ElementRef } from '@angular/core';
import { RouteParams, CanReuse, OnReuse } from '@angular/router-deprecated';
import { SettingsService, TaskService } from '../shared/shared';
import { AnimationBuilder } from '@angular/platform-
browser/src/animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-
browser/src/animate/css_animation_builder';
```

...

Note

Please note the source locations for `AnimationBuilder` and `CssAnimationBuilder`. At the time of writing, the `@angular/animate` barrel has not been registered in any specific bundle within the Angular 2 framework, so we need to use the full path for importing each symbol. This may change in the future so please refer to the official Angular 2 documentation in case you get a 404 error when importing the types.

With all these tools in place, we can begin setting up the foundation for our animation. First let's create a new member in our component controller class under the name of `fadeInAnimationBuilder`. This new class property will represent the `CssAnimationBuilder` instance object that will define the animated transition we are about to build now. First, let's inject the dependencies we need through the class constructor, properly prefixed with access modifiers so they become class members instantly. The code is as follows:

app/timer/timer-widget.component.ts

```
...
export default class TimerWidgetComponent implements OnInit, CanReuse, OnReuse {
  minutes: number;
  seconds: number;
  isPaused: boolean;
  buttonLabelKey: string;
  buttonLabelsMap: any;
  taskName: string;
  fadeInAnimationBuilder: CssAnimationBuilder;

  constructor(
    private settingsService: SettingsService,
    private routeParams: RouteParams,
    private taskService: TaskService,
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {
      this.buttonLabelsMap = settingsService.labelsMap.timer;

      this.fadeInAnimationBuilder = animationBuilder.css();
      this.fadeInAnimationBuilder.setDuration(1000)
        .setDelay(300)
        .setFromStyles({ opacity: 0 })
        .setToStyles({ opacity: 1 });
  }
  // Rest of the class remains the same
}
```

The `AnimationBuilder.css()` method is used in the constructor implementation to instantiate a `CssAnimationBuilder` object, and it is directly assigned to the `fadeInAnimationBuilder` member. Once assigned, we can use the `CssAnimationBuilder` API to define an animation that will kick off after 300 milliseconds, after which will endure along 1000 milliseconds (that is, a second) a style transition on any given DOM element from full transparency to solid color

state. It is worth remarking that the `CssAnimationBuilder` features a chainable API, so we can conveniently chain settings one after another.

But as we pointed out in the beginning of this section, this animation handler is completely agnostic of the elements it can be applied on. Let's see how we can make all this transition happen on an actual DOM element. To do so, we can trigger the animation using the `start()` method exposed by the `CssAnimationBuilder`. This method expects an HTML element in its signature, on which the configured animations will be applied. Go to the `ngOnInit()` hook and add the following block of code at the end:

app/timer/timer-widget.component.ts

```
...
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }

  this.fadeInAnimationBuilder.start(

    this.elementRef.nativeElement.firstElementChild);
}
...
```

As we mentioned, the `start()` method will expect an HTML element on which to apply the animation setup and that we do by feeding the function with the first child node of the `nativeElement` property of the `elementRef` class member. The `ElementRef` type we imported in the constructor gives us a reference pointer to the component directive itself (this is the `PomodoroTimer` component) in the context of the parent view or template it currently exists. Its `nativeElement` property gives us access to the underlying native element of the component, which is usually the template HTML nodes tree. From that point onward, we just need to fetch its `firstElementChild` property value, which will point to the root node of the component template, and apply the animation tweening on it. It is important to remark that the component will not react to any animation applied directly to it. Therefore, we need to traverse the `nativeElement` property after the actual DOM element we want to animate. We can introduce other DOM selectors here, leveraging the web element API methods such as `document.getElementById()` or `document.querySelector()`, but this is discouraged since it creates a tight coupling between the controller and the rendering layers and can compromise future maintainability of our components.

Now, save all your work and re-run the Pomodoro application, accessing the Pomodoro timer. Voila! Our beloved timer now gracefully shows up on screen with a smooth transition.

The CssAnimationBuilder API

We have just seen how we can create agnostic animation handlers with the `CssAnimationBuilder`, and to do so we have leveraged some powerful methods of its API (such as `setDelay`, `setDuration`, `setFromStyles`, or `setToStyles`). This is just a subset of all methods available in its API, which encompass some more methods that are really useful for building complex animations. These methods, including the signatures, are as follows:

- `setDelay(delay: number)`: As we saw in our example, this sets the animation delay and overrides any other animation delay previously defined through CSS.
- `setDuration(duration: number)`: This sets the animation duration and, similar to `setDelay()`, overrides any animation duration previously defined through CSS.
- `setFromStyles(from: {[key: string]: any})`: As we saw in our example, it sets the initial styles for the animation, in the form of a hash object of key/value pairs. Be careful when styling CSS properties named with camel case. All CSS property names must be converted to camel case. In that sense, properties such as margin-top or background-color would turn into `marginTop` or `backgroundColor`.
- `setToStyles(to: {[key: string]: any})`: This sets the destination styles for the animation.
- `setStyles(from: {[key: string]: any}, to: {[key: string]: any})`: This is syntactic sugar to directly access the functionality provided by `setFromStyles` and `setToStyles` in a single method, which obviously sets styles for both the initial state and the destination state.
- `addClass(className: string)`: This adds a class that will remain on the element after the animation has finished. This method is especially useful for overriding CSS properties on DOM elements already managed by CSS transitions.
- `removeClass(className: string)`: As the counterpart of the previous method, this removes a class from the element.
- `addAnimationClass(className: string)`: This adds a temporary class that will be removed at the end of the animation. Angular 2 leverages this method under the covers for handling the CSS hooks triggered by the `ng-animate` class binding we overviewed at the beginning of this chapter.
- `start(element: HTMLElement)`: This starts the animation on the HTML element defined in the payload when executing the method and returns an `Animation` object. This `Animation` object exposes very useful methods we can leverage to implement additional functionalities as callbacks to be executed when the animation is complete, among other functionalities.

All these chainable methods allow us to build really complex animation handlers and reuse them throughout our applications with no effort.

Tracking animation state with the Animation class

Our applications' interactivity does not end in the moment an animation completes its interpolation. In fact, this can become the starting point of many other animations or interactive events occurring in our user interface. For that reason, it is important for our applications to be able to detect when an animation completes its interpolation.

Fortunately, we have the `Animation` class for this, and the `CssAnimationBuilder.start()` method precisely returns an instance of this type, as we can see in the following example:

app/timer/timer-widget.component.ts

```
...
ngOnInit(): void {
  this.resetPomodoro();
  setInterval(() => this.tick(), 1000);

  let taskIndex = parseInt(this.routeParams.get('id'));
  if (!isNaN(taskIndex)) {
    this.taskName = this.taskService.taskStore[taskIndex].name;
  }

  const animation = this.fadeInAnimationBuilder.start(
    this.elementRef.nativeElement.firstElementChild);

  animation.onComplete(() => console.log('Animation completed!'));
}
```

The `Animation` class exposes in its API the `onComplete` event handler, which is fired as soon as the animation triggered by the `CssAnimationBuilder.start()` method finishes. So, we can leverage it to trigger any other action in our app, such as logging operations (as depicted in the preceding example) or further animations.

Regarding the `Animation` class, it is in fact the one that carries out all the hard work of managing the transition. In that sense, the `CssAnimationBuilder` is just a facade providing a friendly interface for setting up the animation flow. When it comes then to performing further operations once the animation is ongoing or has just finished, the `Animation` class is our only resource.

All in all, we will rarely interact with the `Animation` class beyond using its callback functions or leveraging its built-in methods to handle CSS classes or swapping styles when the animation is over. On the other hand, it is not an `Injectable` class so we cannot instantiate it using Angular's dependency injection system. For these reasons, we will not cover its API in detail here.

Developing custom animation directives

We have said several times that one of the advantages of the `CssAnimationBuilder` API is its reusability. Putting together any given animation setup and applying it on not one but many HTML elements later on becomes a breeze. However, directives are the perfect solution when it comes to managing reusability in the Angular arena. So why not get the best of both worlds? As we will see in the next example, wrapping animation within directives becomes the go-to solution for many case scenarios.

Our last animation example in this chapter will introduce a brand new custom directive into our application. The `Highlight` directive leverages the `CssAnimationBuilder` API to change the background color of any given DOM element on the fly, resetting the background color to its original state at the end of the animation. This kind of *flashing* effect has become quite widespread in modern web design for making the user aware that something has just happened on some part of the UI.

Let's start by creating the directive controller class file inside the directives subfolder of our shared features folder, and populate it with the following script. Please note how we keep applying the file naming conventions we embrace and how our directive will be mapped to a CSS class selector this time:

app/shared/directives/highlight.directive.ts

```
import { Directive, ElementRef, OnInit } from '@angular/core';
import { AnimationBuilder } from '@angular/platform-
browser/src/animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-
browser/src/animate/css_animation_builder';

@Directive({
  selector: '.pomodoro-highlight',
  providers: [AnimationBuilder]
})
export default class HighlightDirective {
  cssAnimationBuilder: CssAnimationBuilder;

  constructor(
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {

    this.cssAnimationBuilder = animationBuilder.css()
      .setDuration(300)
      .setToStyles({ backgroundColor: '#fff5a0' });
  }

  ngOnInit() {
    let animation = this.cssAnimationBuilder.start(
      this.elementRef.nativeElement
    );
```

```
      animation.onComplete(() => {
        animation.applyStyles({ backgroundColor: 'inherit' });
      });
    }
}
```

The code is pretty simple in its implementation. We basically build a directive mapped to a CSS class selector whose constructor instantiates a CSS animation consisting of a background-color interpolation to a certain tone of yellow (defined by the `#fff5a0` hex value) along 300 milliseconds. This is our desired *flashy* effect. The `ngOnInit` hook method, which is executed in the very moment that the component affected by this directive is rendered in the view, fires the animation and resets the background color of the affected DOM element back to its original value. As we can see, we are taking advantage of the `applyStyles()` method of the `Animation` class. This method, along with other methods exposed in its API such as `addClasses()` or `removeClasses()` (both expecting an string array with the class names to add or remove, respectively), allows us to interact with the CSS bindings of the animated DOM element.

Before moving on, we need to ensure this new directive is available for the rest of features coexisting in our application, so we need to expose this new directive from the shared facade module as well. The code is as follows:

app/shared/shared.ts

```
import Queueable from './interfaces/queueable';
import Task from './interfaces/task';

import FormattedTimePipe from './pipes/formatted-time.pipe';
import QueuedOnlyPipe from './pipes/queued-only.pipe';

import AuthenticationService from './services/authentication.service';
import SettingsService from './services/settings.service';
import TaskService from './services/task.service';

import RouterOutletDirective from './directives/router-outlet.directive';
import HighlightDirective from './directives/highlight.directive';

const SHARED_PIPES: any[] = [
  FormattedTimePipe,
  QueuedOnlyPipe
];

const SHARED_PROVIDERS: any[] = [
  AuthenticationService,
  SettingsService,
  TaskService
];

const SHARED_DIRECTIVES: any[] = [
  RouterOutletDirective,
  HighlightDirective
];
```

```
export {
  Queueable,
  Task,

  FormattedTimePipe,
  QueuedOnlyPipe,
  SHARED_PIPES,

  AuthenticationService,
  SettingsService,
  TaskService,
  SHARED_PROVIDERS,

  RouterOutletDirective,
  HighlightDirective,
  SHARED_DIRECTIVES
};
```

As you can see in the new refactored facade, the `Highlight` directive is part of the `SHARED_DIRECTIVES` symbol. Therefore, it is available for use on any component already declaring that token in its directives property, such as `TasksComponent`, whose template we are about to tweak now.

Open the component's template and decorate the `ngFor` element with an additional class, as follows:

app/tasks/tasks.component.html

```
<tr *ngFor="let task of tasks; let i = index"
  class="ng-animate highlight">
```

Reload the application and rejoice by watching how our task list flashes upon loading on screen, just to return back to its normal state. Remember that we can apply the same behavior to any other piece of DOM in our application just by importing the directive and binding the class name in the DOM element of our choice. However, you are probably wondering why we built all this boilerplate for delivering just a flashy effect. Wouldn't it be easier to wrap everything around a CSS class perhaps? Well, that is correct... unless you want to interact with the animation, and that is what we are going to do next.

Interacting with our directive from the template

In fairness, having a directive triggering an animation like this makes no sense, but it would be great if we could interact with the animation. Moreover, if we could actually interact right from the template. To do so, we can assign an exportable token name to our directive so we can refer to it from the same element intervened by the directive. Do you remember how we used to refer to the ngForm directive when handling forms? Here, we take advantage of the same technique, as we will see later. First, proceed to update the Highlight directive by adding a new property to the directive setup named exportAs, with the value highlight. The value defined there will become the name we should refer to when trying to access the directive API from within outside. How shall we do this? A little bit of patience, first let's ditch the ngOnInit method by changing its name to colorize, thereby removing the type from the first line of imports and the interface implementation from the class. The code is as follows:

app/shared/directives/highlight.directive.ts

```
import { Directive, ElementRef } from '@angular/core';
import { AnimationBuilder } from '@angular/platform-
browser/src/animate/animation_builder';
import { CssAnimationBuilder } from '@angular/platform-
browser/src/animate/css_animation_builder';

@Directive({
  selector: '.pomodoro-highlight',
  providers: [AnimationBuilder],
  exportAs: 'pomodoroHighlight'
})
export default class HighlightDirective {
  cssAnimationBuilder: CssAnimationBuilder;

  constructor(
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) {

    this.cssAnimationBuilder = animationBuilder.css()
      .setDuration(300)
      .setToStyles({ backgroundColor: '#fff5a0' });
  }

  colorize() {
    let animation = this.cssAnimationBuilder.start(
      this.elementRef.nativeElement
    );

    animation.onComplete(() => {
      animation.applyStyles({ backgroundColor: 'inherit' });
    });
  }
}
```

Rerun the example. Now the table does not flash upon loading, which is fine. With all this in

place, it's time to update our template. First, we will add a local reference named row (or whatever name you fancy) in the same HTML node impacted by the directive, pointing to highlight which is, as we now know, the public name of our directive. Same as we used to do when referencing the state and validity of our forms with ngForm, now we can access the directive public API, which exposes the colorize() method we just created out of the former ngOnInit interface method. We can safely execute that method now just by pointing to the local template reference, like this:

app/tasks/tasks.component.html

```
<tr *ngFor="let task of tasks; let i = index"
    class="ng-animate pomodoro-highlight"
    #row="pomodoroHighlight"
    (click)="row.colorize()">
```

Reload the application and click on any task, queuing it up and off. Its row will flash momentarily.

Tip

In the implementation of the Highlight directive, we hardcoded the CSS value in body of the directive. In order to make this directive more sustainable and scalable, we should prevent this approach in larger applications. As a personal exercise, we would suggest you to refactor the directive to use an attribute selector, such as [pomodoroHighlight], whose value is parsed by a class member annotated with the @Input decorator, so you can configure custom flashing colors when binding the directive in your component templates.

Looking into the future with ngAnimate 2.0

Most of the procedures we have seen along this chapter inherit the animation logic already present in Angular 1.x, which was based on attaching CSS classes and relying on `keyframe` animations previously created in our style sheets. Unfortunately, this approach falls short when it comes to performance optimization, implementing concurrent animations or performing advanced interactions, like the ones proposed in Material Design, just to name a few shortcomings.

This is why the Angular team is now working on a new set of modules that will add an unparalleled layer of performance and abstraction through the use of a **domain specific language** (**DSL**) with full programmatic API. This project will take form in what is known as **ngAnimate 2.0**, and it will be released at some point later in 2016. The approach proposes a programmatic system that returns to JavaScript, tapping into CSS and allowing for programmatic control all around.

In a nutshell, some of the great improvements that ngAnimate 2.0 will feature are as follows:

- Use an animation factory to create animations through a programmatic API with support for concurrent and sequential animations
- Staggering animations handled 100 percent through JavaScript, bringing the same logic of CSS `Keyframes` animation to JavaScript, compounded by universal sequencing and animation chaining
- Performance tuning with dramatically fewer reflows
- Multi DOM-level animations and a sound control over CSS
- Better event integration and full control over the animation flow, with the ability to fast-forward or reverse an animation interpolation programmatically
- Solid support for Material Design and improved handling of user interaction events, mouse clicks, and so on
- Adaptive styling
- Full testability and support for animation asserts in our test specs

These and many more features will become available as soon as ngAnimate 2.0 is released. At the time of writing, the API is not public yet as ngAnimate 2.0 is a proof of concept, so providing a deeper coverage of its mechanism is out of the scope of this book. However, keep an eye on future announcements, since ngAnimate 2.0 will become an important milestone in the way animations are handled in JavaScript.

Summary

In this chapter, we discussed some of the different techniques currently available for handling animations in Angular 2. First, we looked at the basic classname-based animation with CSS3 transitions with the help of the Angular built-in directives. Then, we saw how Angular provides class-based animation helpers out of the box, so we can easily implement our own transitions by following some basic conventions in our style sheets. Then, we discussed programmatically-managed animation development with the amazing animation builders currently existing in Angular 2, with some examples of how to take advantage of the `Animation` class callbacks to trigger actions along the animation lifecycle.

The last leg of this chapter was devoted to getting some insights about ngAnimate 2.0, which is set out to take over the previous techniques in the near future.

Our next and final chapter in the book will sum up all we have learned so far by introducing one of the most relevant requirements in modern web application development: unit testing. Angular 2 embraces common industry standards and libraries, while introducing some particular tools to turn unit testing into an enjoyable task.

Chapter 10. Unit testing in Angular 2

The hard work of the previous chapters has materialized into a working application we can be proud of. But how can we ensure a painless maintainability in the future? A comprehensive automated testing layer will become our lifeline once our application begins to scale up and we have to mitigate the impact of bugs caused by new functionalities colliding with the already existing ones.

Testing (and more specifically unit testing) is meant to be carried out by the developer as the project is being developed. However, we will cover all the intricacies of testing Angular 2 modules in brief in this chapter, now that the project is in a mature stage.

In this chapter, you will see how to implement testing tools to perform proper unit testing of your application classes and components.

In this chapter we will:

- Look at the importance of testing and, more specifically, unit testing
- Review the different parts of a JavaScript unit test
- Discover Jasmine, our testing framework of choice
- Learn how to set up a unit testing environment with Jasmine and SystemJS
- Build a test spec testing a pipe
- Design unit tests for components, with or without dependencies
- Put our routes to the test
- Implement tests for services, mocking dependencies, and stubs
- Intercept XHR requests and provide mocked responses for refined control
- Discover how to test directives as components with no view
- Introduce other concepts and tools such as Karma, code coverage tools, or E2E testing

Why do we need tests?

What is a unit test? If you're familiar already with unit testing and test-driven development, you can safely skip to the next section. If not, let's say that unit tests are part of an engineering philosophy that takes a stand for efficient and agile development processes by adding an additional layer of automated testing on the code before it is developed. This is the core concept is that each piece of code is delivered with its own test, and both pieces of code are built by the developer who is working on that code. First we design the test against the module we want to deliver, checking the accuracy of its output and behavior. Since the module is still not implemented, the test will fail. Hence, our job is to build the module in such a way that it passes its own test.

Unit testing is quite controversial. While there is a common agreement about how beneficial test-driven development for ensuring code quality and maintenance upon time is, not everybody undertakes unit testing in their daily practice. Why is that? Well, building tests while we develop our code can feel like a burden sometimes, particularly when the test winds up being bigger in size than the piece of functionality it aims to test.

However, the arguments favoring 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. In that sense, if we try to test an existing piece of code and we find ourselves blocked at some point, chances are that the piece of code we aim to test is not well designed and shows off a convoluted interface that requires some rethinking. On the other hand, building testable modules can help to early detect side effects on other modules.
- Refactoring tested code is the lifeline against introducing bugs in later stages. Any development is meant to evolve with time, and on every refactor the risk of introducing a bug that will only pop up in another part of our application is high. Unit tests are a good way to ensure that we catch bugs in an early stage, either when introducing new features or when updating existing ones.
- Building tests is a good way to document our code APIs and functionalities. And this becomes a priceless resource when someone not acquainted with the codebase 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 see the overall form of a test.

Parts of a unit test in Angular 2

There are many different ways to test a piece of code, but we will focus on how Angular 2 aims to test modules. The first thing we need is a test framework, providing utility functions for building *test suites* containing one or several *test specs* each.

```
describe('The submit component', () => { // Test suite

  it('should be rendered disabled by default', () => { // Test spec
    // ... Test spec implementation goes here
  });

});
```

Each *test spec* checks out a specific functionality of the feature described in the suite description argument, and declares one or several *expectations* in its body. Each *expectation* takes a value, which we call the *expected* value, and is compared against an *actual* value by means of a *matcher* function, which checks whether expected and actual values match accordingly. This is what we call an *assertion*, and the test framework will pass or fail the spec depending on the result of such assertion. The code is as follows:

```
describe('The submit component', () => { // Test suite

  it('should be rendered disabled by default', () => { // Test spec

    // Test assertion based on an instance of the submit component
    expect(submitComponent.disabled).toBe(true);

  });

});
```

In the previous example, `submitComponentInstance.disabled` will return the *actual* value that is supposed to match the *expected* value declared in the `toBe()` matcher function.

Dependency injection in unit tests

It is quite common to perform several operations before executing each spec: instantiating the component or service class we want to test, fetching a dependency, declaring a mock argument, and so on. The most common operation when testing Angular 2 modules is to override the default providers of the injector with the `beforeEachProviders()` function. This function must be executed *before* executing anything else and takes this shape:

```
describe('The submit component', () => {

  beforeEachProviders(() => [
    TestComponentBuilder,
    MyCustomService,
    provide(Router, { useClass: RootRouter })
  ]);

  it('should be rendered disabled by default', () => {
    expect(submitComponent.disabled).toBe(true);
  });

});
```

In this example and prior to executing any spec, we are setting up a list of DI providers we need our test injector to be aware of. As we can see, we can just declare class tokens or even override dependencies by binding a token to a class, value, or factory function of our choice with the `provide()` function. In fact, replacing module dependencies by mock types with fake data or functionality is indeed a quite common practice when testing.

We're still not done in regards of DI: what about instantiating the objects we require for testing or resetting the fixture information we need for each test? The `beforeEach()` function is the natural place for fetching the dependencies previously declared in `beforeEachProviders()`, instantiating the classes we need to test, or performing any operation required for preparing the objects or data fixtures our test specs need. It must be executed *after* the `beforeEachProviders()` and *before* the test specs. In order to do so, it leverages the `inject()` function.

In the following piece of code, we strip down up a bit the example provided and introduce all the necessary steps. Please pay attention to the inline code comments, since they describe the intent of each block of code:

```
describe('The submit component', () => {

  // We declare the dependencies we need the provider to manage
  beforeEachProviders(() => [
    TestComponentBuilder
  ]);

  // A variable will allocate the test component builder
  let testComponentBuilder: TestComponentBuilder;
```

```
  // Before each spec we fetch an instance of
  // TestComponentBuilder right from the injector
  beforeEach(inject([TestComponentBuilder],
                     (tcb: TestComponentBuilder) => {
      testComponentBuilder = tcb;
    })
  );

  // The done function argument configures our spec as asynchronous
  it('should be rendered disabled by default', done => {

    // The test component builder asynchronously resolves to a
    // fixture object wrapping an actual instance of SubmitComponent
    testComponentBuilder
      .createAsync(SubmitComponent)
      .then(testComponent => {

        // We fetch the component instance
        // out from the fixture wrapper
        const submitComponent = testComponent.componentInstance;

        // We evaluate the assertion with a given matcher function
        expect(submitComponent.disabled).toBe(true);

        // finally, the done() function resolves
        //the asynchronous spec
        done();
      });
  });
});
```

Do not worry about the `TestComponentBuilder`, we will cover it shortly. Pay special attention to the way we get an instance of the `TestComponentBuilder` type by passing the class token to the `inject()` function, which will return in its callback argument the desired instance that we will bind to a variable in the outer scope for later use in our specs. A more expressive way to fetch object instances through the injector would be to fetch an actual `Injector` object instance and then leverage its `get()` helper method:

```
let myCustomService; // A custom service of ours

beforeEach(inject([Injector], (injector: Injector) => {
    myCustomService = injector.get(MyCustomService);
  })
);
```

Ultimately, choose the syntax you find more convenient. The rest of the example basically entails the common operations required when testing components: creating a testing component fixture, fetching an actual component instance from the fixture wrapper, conducting assertions on the component instance, and finally resolving the asynchronous *spec* with the `done()` command. We will see all these in detail once we overview component testing later on in this chapter.

There are more elements in a unit test (*mocks*, *spies*, and so on), but we will see the most

common suspects along the following pages.

There are two important questions: how do all these tests are executed and where do we check the pass/fail results? For the first question, let's just say we will need a testing framework for JavaScript, and at the time of this writing, Jasmine (http://jasmine.github.io) seems to be the preferred testing framework in the Angular team. Regarding the second question, we will need a *spec runner*, and Jasmine precisely provides an HTML-based runner out-of-the-box, so let's see how we can configure Jasmine to do this.

Setting up our test environment

Our first step will be to download and integrate Jasmine in our project. To do so, go to the console in our project and execute the following `npm` command that will install the `jasmine-core` package (including the exact versions of the dependencies required rather than using npm's default semver range operator, hence the `--save-exact` flag) required for our tests:

```
$ npm install jasmine-core --save-dev --save-exact
```

With the package installed, we need to ensure that our project contains the proper typings for the Jasmine global objects and function matchers, so the compiler can recognize the types and thus build our tests:

```
$ typings install jasmine --save --ambient
```

With all the libraries in place, let's see how we will put together a spec runner to process our tests and output results.

Implementing our test runner

We are going to run our tests in a browser, and in order to do so we will need to set some base testing providers that are specific to the browser platform. Create a folder named `test` at the root of our project and save the following file there:

test/setup.ts

```
import { resetBaseTestProviders, setBaseTestProviders } from
'@angular/core/testing';
import { BROWSER_APP_DYNAMIC_PROVIDERS } from "@angular/platform-browser-
dynamic";
import {
  TEST_BROWSER_STATIC_PLATFORM_PROVIDERS,
  ADDITIONAL_TEST_BROWSER_PROVIDERS
} from '@angular/platform-browser/testing';

resetBaseTestProviders();

setBaseTestProviders(
  TEST_BROWSER_STATIC_PLATFORM_PROVIDERS,
  [
    BROWSER_APP_DYNAMIC_PROVIDERS,
    ADDITIONAL_TEST_BROWSER_PROVIDERS
  ]
);
```

We will import this file into our testing implementation shortly and there is no need to cover it in detail at this point. Let's just say these providers contain DOM adapters required to perform certain operations in Shadow DOM implementations.

Before moving on, it is important to highlight one of the conventions used in this book for Angular 2 unit tests: file naming and spec location. There is a general agreement on saving our test spec files in the same location where the tested module lives. In order to make things even clearer, we will name the spec files after the name of the code unit they test, and will append the `.spec` suffix to the filename. This way, it becomes easier to locate the tests corresponding to each module, check what is tested and what modules lack a test, and get better acquainted with the code base. Keep in mind that a good test becomes a valuable piece of documentation by itself.

Let's see this naming convention in action by creating a temporary test spec at the root of your project, so we can check whether our runner is working fine:

test.spec.ts

```
describe('Our test runner', () => {
  it('is alive!', () => {
    expect(true).toBe(true);
  });
});
```

Let's create our spec runner file now at the root of our project. The spec runner is pretty similar to the main HTML page in regards to the script resources required but also adds on top of that all the required include scripts from Jasmine and the testing bundle from Angular 2. The following implementation also declares an array with string pointers to the locations of the browser testing setup file and the proof of concept test we just built, besides featuring a variable storing the path to the compiled TypeScript files:

spec-runner.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="content-type"
      content="text/html;charset=utf-8">
    <title>Pomodoro App Unit Tests</title>

    <link rel="stylesheet"
      href="node_modules/jasmine-core/lib/jasmine-core/jasmine.css">
    <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine.js"></script>
    <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine-html.js">
</script>
    <script src="node_modules/jasmine-core/lib/jasmine-core/boot.js">
    </script>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>

    <script src="systemjs.config.js"></script>
  </head>
  <body>
    <script>
      // Your typescript compiler 'outDir' parameter value
      var outDir = 'built';

      // Enlist your specs in the following spec collection array,
      // next to the setup import file, all with no file extension
      var specCollection = [
        'test/setup',
        'test.spec' // The test spec we just built
      ];

      // We load all specs asynchronously from the built folder
      // and evaluate their output at once
      Promise.all(specCollection.map(specPath => {
        return System.import(`${outDir}/${specPath}`);
      }))
      .then(window.onload)
      .catch(console.error.bind(console));
    </script>
  </body>
```

```
</html>
```

Check the import scripts and focus on the last block of code. As we said, we define the path to the compiled files (according to our `tsconfig.json` setup) and an array of string paths to the main browser testing setup and our specs, which is only one at this moment. The `System.config` implementation is pretty straightforward and reminds of the one we already have for launching our project at `index.html`. Our last block is a bit more complex and requires some more attention. Basically, we create an array of `System.import` commands by mapping the setup and spec paths array into another array that combines those paths with the `outDir` folder location. Each `System.import()` execution returns a promise, so the resulting array can be executed altogether through a `Promise.all()` command, triggering a window `onload` even upon resolving. It is precisely this event that Jasmine is waiting for to render the pass/fail report.

Time to see all this in action! Save the changes, run the compiler to have transpiled versions of the the test setup file and our just created spec, and then run the local web server by browsing to the spec-runner file URL in the browser location bar. You should see a web report like this:
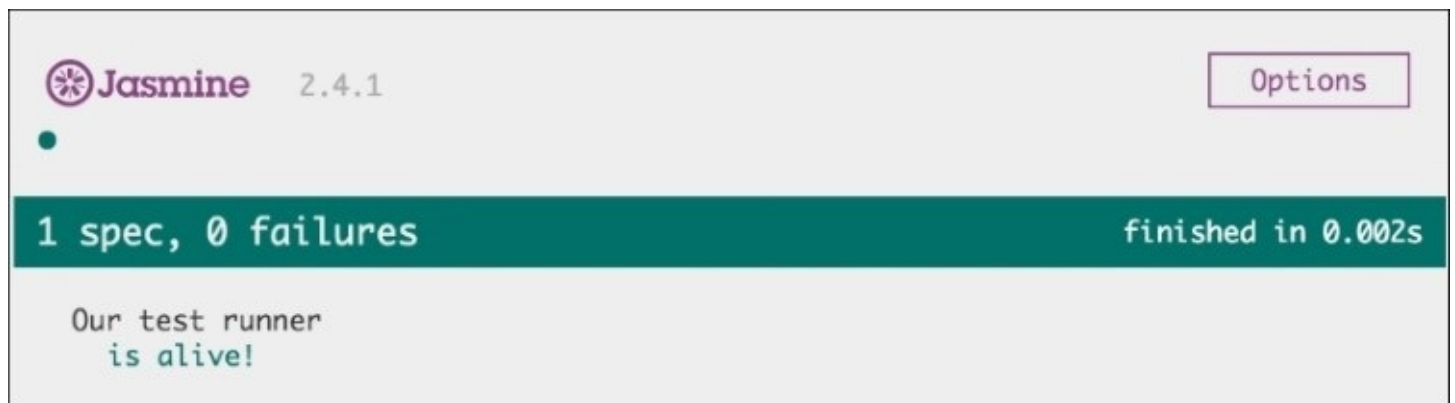
Setting up NPM commands

Testing our modules is an iterative process, so we can ease things a bit in order to make the whole flow smoother. To do so, we can set up some wrapping commands around the common tasks of erasing the contents of the build folder, recompiling the project and triggering a web server pointing to our test runner. As a matter of fact, the package.json file can allocate a test command, which will also trigger pretest and posttest scripts when executed. With this knowledge in hand, let's update the scripts block in our package.json file to include commands that perform all the aforementioned operations:

package.json

```
...
"scripts": {
  "postinstall": "npm run typings install",
  "tsc": "tsc",
  "tsc:w": "tsc -w",
  "lite": "lite-server",
  "prestart": "tsc",
  "start": "concurrently \"npm run tsc:w\" \"npm run lite\" ",
  "typings": "typings",
  "pretest": "rm -rf ./built && npm run tsc",
  "test": "npm run lite --open=/spec-runner.html"
},
...
```

Whenever you want to run the whole batch of tests on our application, just go to the console and run npm test at the project location prompt. Remember that the compiler is not executed in watch mode, so during your development sessions it is safe to run npm start instead and load the spec runner in a separate browser window to check the evolution of our tests, if required.

We're done with our setup! In the following pages, we will go through different examples of tests structures for components, directives, pipes, services and routes, taking some modules of the pomodoro app project as an example. There is no one-size-fits-all pattern when unit testing Angular 2 modules, but different approaches depending on the subject of test. All in all, we will go finding recurring patterns in our tests and the rest of the implementation will explain by itself, so we will not get into much detail describing what the code does in each example. Ultimately, that is what TDD is all about: explaining how code works by putting it to the test.

Angular 2 custom matcher functions

What kind of checks can we perform with matcher functions in Angular 2? Well, the answer is: quite a lot. Besides the most common matchers such as to `toBe()` or `toEqual()`, we can use any built-in Jasmine matcher. Please refer to the Jasmine official site for a complete rundown on the matchers available. On top of that, Angular 2 implements a set of custom matchers to perform common operations when testing Angular 2 specific modules:

- `toBePromise()`: This expects the value to be a Promise
- `toBeAnInstanceOf(expected: any)`: This expects the actual value to be an instance of a class defined in the `expected` argument
- `toHaveText(expected: any)`: This expects the element to have exactly the text defined in the `expected` argument
- `toHaveCssClass(expected: any)`: This expects the element to have the CSS class defined in the `expected` argument
- `toHaveCssStyle(expected: any)`: This expects the element to be styled with the given CSS styles defined in the payload
- `toImplement(expected: any)`: This expects a class to implement the interface of the given class
- `toContainError(expected: any)`: This expects an exception to contain an expected error text
- `toThrowErrorWith(expectedMessage: any)`: This expects a function to throw an error with the given error text when executed
- `toMatchPattern(expectedMessage: any)`: This expects a string to match the given regular expression

All the preceding matcher functions resolve to a Boolean value. Sometimes, we will want to evaluate in our assertion the opposite of the comparison actually implemented in the matcher function. For those cases we just need to prepend a `.not` modifier before the matcher:

```
expect(true).not.toBe(false);
```

Testing pipes

A pipe is basically a class that implements the `PipeTransform` interface, hence exposing a `transform()` method that is usually synchronous. In that sense, pipes are the perfect candidates for taking our first steps in the world of unit testing Angular 2 modules. We will begin then by testing `FormattedTimePipe`, creating as we mentioned a test spec right next to its code unit file. The code is as follows:

app/shared/pipes/formatted-time.pipe.spec.ts

```
import FormattedTimePipe from './formatted-time.pipe';
import {
  describe,
  expect,
  it,
  beforeEach} from '@angular/core/testing';

describe('shared:FormattedTimePipe', () => {
  let formattedTimePipe: FormattedTimePipe;

  beforeEach(() => formattedTimePipe = new FormattedTimePipe());

  // Specs with assertions
  it('should expose a transform() method', () => {
    expect(typeof formattedTimePipe.transform).toEqual('function');
  });

  it('should transform 50 into "0h:50m"', () => {
    expect(formattedTimePipe.transform(50)).toEqual('0h:50m');
  });

  it('should transform 75 into "1h:15m"', () => {
    expect(formattedTimePipe.transform(75)).toEqual('1h:15m');
  });

  it('should transform 100 into "1h:40m"', () => {
    expect(formattedTimePipe.transform(100)).toEqual('1h:40m');
  });

});
```

We import the pipe token in order to instantiate it and bind it to a variable before running each test, which will grab this variable and introspect its type or will pass different values to its transform method to check whether we obtain the expected value or not.

What about the testing cycle we mentioned earlier? In a normal scenario, we would add our test first, leveraging the test itself to design our pipe interface. Our test would fail at first and then we would develop its implementation. When defining our assertions, we should run the extra mile and define assertions where our pipe has to confront wrong inputs or unexpected scenarios. As our code evolves and is refactored, our tests will have to be refactored and

simplified as well.

Note

It is worth remarking that we are importing the Jasmine global functions from the Angular 2 testing bundle, and not from Jasmine itself. This is because Angular 2 overrides Jasmine's built-in functions, although the functionality and interface remains the same.

Save the file and declare it in our `specCollection` array at our spec runner (you can safely remove the reference to the test spec since we will no longer use it):

spec-runner.html

```
...
var specCollection = [
  'test/setup',
  'app/shared/pipes/formatted-time.pipe.spec'
];
...
```

In the next sections, we will see how to create different tests.

Note

Every time we create a new test spec, you will have to append its path (without the file extension) to the `specCollection` array variable at `spec-runner.html`. *Do not forget this*, since we will not make explicit reference to this requirement from now on. Failing to include the reference will turn into non-execution of the test. Therefore, the spec will not show up in the spec runner report.

Testing components

Testing pipes is pretty straightforward but testing components can become a more daunting experience when approached for the first time. There are too many questions: how can we test a component that needs to be bootstrapped somewhere? Good news is that Angular 2, and more specifically its testing bundle, contains a class named `TestComponentBuilder` that can be used to instantiate fully functional components of any given type, wrapped by a fixture object that gives us access to the component instance object or its compiled HTML view. In summary, any instance of the `TestComponentBuilder` exposes the following properties and methods:

- `debugElement`: This is the `DebugElement` associated with the root element of this component.
- `ComponentInstance`: This returns the instance object of the root component class, with full access to all its properties and methods.
- `NativeElement`: This returns the native element at the root of the component.
- `DetectChanges()`: This triggers a change detection cycle for the component. We want to run this method in order to check that the changes occurred on the component state should we update any of its properties or execute its methods.
- `destroy()`: This triggers component destruction.

With all these points in mind, let's create our first component test. `TaskIconsComponent` is a perfect candidate to start with:

app/tasks/task-icons.component.spec.ts

```
import TaskIconsComponent from './task-icons.component';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import { TestComponentBuilder } from '@angular/compiler/testing';

describe('tasks:TaskIconsComponent', () => {
  let testComponentBuilder: TestComponentBuilder;

  // First we setup the injector with providers for our component
  // and for a fixture component builder
  beforeEachProviders(() => [TestComponentBuilder]);

  // We reinstantiate the fixture component builder
  // before each test
  beforeEach(
    inject([TestComponentBuilder],
    (_testComponentBuilder: TestComponentBuilder) => {
      testComponentBuilder = _testComponentBuilder;
    }
```

```
  ));

  // Specs with assertions
  it('renders 1 image for each pomodoro session required', done => {
    // We create a test component fixture on runtime
    // out from the TaskIconsComponent symbol
    testComponentBuilder
    .createAsync(TaskIconsComponent)
    .then(componentFixture => {

      // We fetch instances of the component and the rendered DOM
      let taskIconsComponent = componentFixture.componentInstance;
      let nativeElement = componentFixture.nativeElement;

      // We set a test value to the @Input property
      // and trigger change detection
      taskIconsComponent.task = { pomodorosRequired: 3 };
      componentFixture.detectChanges();

      // these assertions evaluate the component's surface DOM
      expect(nativeElement.querySelectorAll('img').length).toBe(3);

      // We finally destroy the component fixture and
      // resolve the async test
      componentFixture.destroy();
      done();

    })
    .catch(e => done.fail(e));
  });

});
```

Take a minute to look at the import statement block at the top of the file. Besides the basic testing functions, we are importing the symbols pertaining to the DI-related functions of the testing bundle, apart from the setup methods beforeEach and beforeEachProviders. First, we will execute beforeEachProviders, passing as an argument a lambda function returning the array of providers we will need. Then, the beforeEach function uses the injector to fetch an object instance of type TestComponentBuilder, which we previously declared as a provider, and binds it to the testComponentBuilder variable. Inside the test spec, we use object variable to execute the asynchronous *promisified* createAsync() method, which will return a fixture around our component of choice (defined as an argument). As we saw already at the beginning of this section, we can inspect the fixture to grab an actual instance of TaskIconsComponent and its underlying native element.

Then, we begin interacting with the component by configuring its properties. Every time we update any input property or execute a method that might change the component state, we need to execute the fixture's detectChanges() method to trigger change detection and hence reflect that state change in the nativeElement. This allows us to test assertions using a matcher function to compare the amount of DOM nodes generated against the expected amount of nodes configured in the matcher. Finally, we destroy the component and resolve the

asynchronous function spec we're in.

Usually, a test has more than one spec, one per each functionality described, for instance. Thus, let's add another it statement right after the one we already have inside the `describe()` suite:

app/tasks/task-icons.component.spec.ts (continued)

```
...
it('should render each image with the proper width', done => {
  testComponentBuilder.createAsync(TaskIconsComponent)
  .then(componentFixture => {
    let taskIconsComponent = componentFixture.componentInstance;
    let nativeElement = componentFixture.nativeElement;
    let actualWidth;

    taskIconsComponent.task = { pomodorosRequired: 2 };
    taskIconsComponent.size = 60;
    componentFixture.detectChanges();

    actualWidth = nativeElement
      .querySelector('img')
      .getAttribute('width');

    expect(actualWidth).toBe('60');
    done();
  })
  .catch(e => done.fail(e));
});
...
```

We can add as many specs as we feel necessary to provide a broad coverage of all scenarios.

Tip

Debugging our own tests

As we create more and more specs, chances are we will introduce some bugs in our own test implementations, turning this into a general failure in our test report. Tracking down these issues can be a bit tricky, so the best way to address this scenario is by isolating test suites or specs execution or, all the way around, disabling the execution of broken tests temporarily. To do so, we can use variations of the `describe()` and `it()` functions, by prepending a letter to the function name, as follows:

- `fdescribe()`: This instructs the test runner to only run the test cases in this group. It can be used as `ddescribe()` as well.
- `fit()`: The test runner will only execute this test, disregarding all the others. It can be used as `iit()` as well.
- `xdescribe()`: This instructs the runner to exclude this test suite from execution.
- `xit()`: This instructs the runner to exclude this test spec from execution.

Testing components with dependencies

In the previous section, we undertook our very first unit test, taking a simple component with no dependencies as an example. However, components and other Angular 2 modules usually have dependencies injected. The unit test needs to reflect this circumstance. Let's look at `TimerWidgetComponent` as an example. This tiny component requires all these dependencies injected through its constructor:

app/timer/timer-widget.component.ts

```
...
constructor(
    private settingsService: SettingsService,
    private routeParams: RouteParams,
    private taskService: TaskService,
    private animationBuilder: AnimationBuilder,
    private elementRef: ElementRef) { ...
```

Thus, in order to instantiate the component within the fixture returned by the `TestComponentBuilder` factory, we need to declare the providers for these dependencies at the test injector and have them injected somehow. On top of that, the component had some important nuances in its execution: it is sensitive to URL params and one of its dependencies (`TaskService`) performs underlying XHR operations by means of the `Http` module. It needs to be intercepted and properly mocked.

This might sound quite daunting, but the truth is that you already know all the code procedures required to put together this test. Let's see it with inline comments in the code:

app/timer/timer-widget.component.test.ts

```
import TimerWidgetComponent from './timer-widget.component';
import { provide } from '@angular/core';
import { RouteParams } from '@angular/router-deprecated';
import { SettingsService, TaskService } from '../shared/shared';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders,
  setBaseTestProviders } from '@angular/core/testing';
import { TestComponentBuilder } from '@angular/compiler/testing';
import { Http, BaseRequestOptions }  from '@angular/http';
import { MockBackend } from '@angular/http/testing';
import 'rxjs/add/operator/map';

describe('timer:TimerWidgetComponent', () => {
  let testComponentBuilder: TestComponentBuilder;
  let componentFixture: any;
```

```
  // First we setup the injector with providers for our component
  // dependencies and for a fixture component builder
  // Note: Animation providers are not necessary
  beforeEachProviders(() => [
    TestComponentBuilder,
    SettingsService,
    TaskService,

    // RouteParams is instantiated with custom values upon injecting
    provide(RouteParams, {useValue: new RouteParams({id: null})}),

    // We replace the Http provider injected later in TaskService
    MockBackend,
    BaseRequestOptions,
    provide(Http, { useFactory:
      (backend:MockBackend, options:BaseRequestOptions) => {
        return new Http(backend, options);
      },
      deps: [MockBackend, BaseRequestOptions]
    }),
    TimerWidgetComponent
  ]);

  // We reinstantiate the fixture component builder before each test
  beforeEach(inject([TestComponentBuilder],
    (_testComponentBuilder: TestComponentBuilder) => {
      testComponentBuilder = _testComponentBuilder;
    }
  ));
});
```

You have probably noticed there is not a single test assertion in the suite. We will get there in a minute, but now let's overview each piece of code in the script. The test suite implementation contains everything you already know about injecting providers in tests. First we use `beforeEachProviders()` to declare all the providers we need the injector to be aware of. As you will remember, the `TimerWidgetComponent` had a dependency on `RouteParams`, which allowed us to fetch the value of the `id` query string parameter, if any. Obviously, we not only need to inject that provider, but our injector ought to return an instance of it with the parameter properly populated for our testing purposes:

```
provide(RouteParams, {useValue: new RouteParams({id: null})}),
```

A bit more attention is required to understand how we accomplish the HTTP requests performed by `TaskService`. The default constructor of the `Http` module requires a backend connection object implementing the `ConnectionBackend` interface. For our test, we need to provide the injector with a working version of `Http` and thus we leverage `MockBackend`, which implements the interface required. Later in this chapter, we will see how we can leverage this class to intercept XHR requests and return canned responses for our tests.

```
BaseRequestOptions,
provide(Http, { useFactory:
  (backend:MockBackend, options:BaseRequestOptions) => {
    return new Http(backend, options);
```

```
    },
    deps: [MockBackend, BaseRequestOptions]
}),
```

With all the providers properly available from the injector, we can leverage it to instantiate a new `TestComponentBuilder` factory object before executing any test.

```
beforeEach(inject([TestComponentBuilder],
  (_testComponentBuilder: TestComponentBuilder) => {
    testComponentBuilder = _testComponentBuilder;
  }
));
```

With all the testing scaffolded and ready, let's introduce our first test spec. Append the following piece of code right after the `beforeEach(...)` block within the body of the `describe(...)` suite:

```
it('should initialise with the pomodoro counter at 24:59', done => {
  // We create a test component fixture on
  // runtime out from the component symbol
  testComponentBuilder
  .createAsync(TimerWidgetComponent)
  .then(componentFixture => {

    // We fetch instances of the component and the rendered DOM
    let timerWidgetComponent = componentFixture.componentInstance;
    let nativeElement = componentFixture.nativeElement;

    // We execute the OnInit hook and trigger change detection
    timerWidgetComponent.ngOnInit();
    componentFixture.detectChanges();

    // These assertions evaluate the component properties
    expect(timerWidgetComponent.isPaused).toBeTruthy();
    expect(timerWidgetComponent.minutes).toEqual(24);
    expect(timerWidgetComponent.seconds).toEqual(59);

    componentFixture.destroy();
    done(); // Resolve async text
  })
    .catch(e => done.fail(e));
});
```

As you can see, our newly created spec executes seamlessly by instantiating a component fixture wrapping the component instance and native element we require. We execute the component's `ngOnInit()` hook method to force its initialization as if had been rendered on a view. Then, we trigger the fixture's `detectChanges()` method that will trigger a change detection cycle on our component instance, applying any state change as a result of the operations taken place within `ngOnInit()`.

The following spec, which you can append to the `describe()` body right after the previous test spec, reinforces these concepts:

```
it('should initialise displaying the default labels', done => {
  testComponentBuilder
    .createAsync(TimerWidgetComponent)
    .then(componentFixture => {
      componentFixture.componentInstance.ngOnInit();
      componentFixture.detectChanges();

      expect(componentFixture.componentInstance.buttonLabelKey)
        .toEqual('start');

      expect(componentFixture.nativeElement
        .querySelector('button')
        .innerHTML.trim())
        .toEqual('Start Timer');

      componentFixture.destroy();
      done();
    })
    .catch(e => done.fail(e));
});
```

Overriding component dependencies for refined testing

In the previous examples, we saw how we could declare and inject the providers that our subjects of testing required. We also saw how we could leverage the `provide()` function to pass the injector an instance of any given provider already populated with the values we require. In that sense, `provide()` is not just used to replace dependency types upon injecting providers, but to customize the way we want a particular provider of that specific type to be injected.

However, can we override providers at a test spec level? The answer is yes, and it is quite useful when it comes to mock dependency values for certain tests. In our next test spec, we will continue testing the timer widget component. However, we will override the `TaskService` provider this time, replacing it by an object literal with mock data. We will also override the default `RouteParams` injection with another instance object of `RouteParams`, featuring an actual value for the `id` parameter.

Add this test spec right after the previous two specs within the body of the `describe(...)` function:

```
it('should initialise displaying a specific task', done => {
  // We mock the TaskService provider with some fake data
  let mockTaskService = {
    taskStore: [{
        name: 'Task A'
      }, {
        name: 'Task B'
      }, {
        name: 'Task C'
      }
    ]
  };

  testComponentBuilder
  .overrideProviders(TimerWidgetComponent, [
    provide(RouteParams, { useValue: new RouteParams({ id: '1' }) }),
    provide(TaskService, { useValue: mockTaskService })
  ])
  .createAsync(TimerWidgetComponent)
  .then(componentFixture => {
    componentFixture.componentInstance.ngOnInit();
    componentFixture.detectChanges();

    expect(componentFixture.componentInstance.taskName)
      .toEqual('Task B');

    expect(componentFixture.nativeElement.querySelector('small'))
      .toHaveText('Task B');

    componentFixture.destroy();
    done();
  })
```

```
    .catch(e => done.fail(e));
});
```

The code is pretty self-explanatory, but let's take some minutes to analyze this block:

```
testComponentBuilder.overrideProviders(TimerWidgetComponent, [
  provide(RouteParams, { useValue: new RouteParams({ id: '1' }) }),
  provide(TaskService, { useValue: mockTaskService })
])
```

Basically, we leverage the `overrideProviders()` method of the `TestComponentBuilder` factory, which will expect in its first argument the type of the component whose providers we want to override and an array of providers as a second argument. We can insert in such an array any kind of type override or replacement by means of the `provide()` function.

In order to get the `overrideProviders()` to work, it parses the current `providers` property of the component decorator whose providers we want to override. If the component does not feature the property in its decorator (mostly because all its dependencies are inherited from the root injector), Angular will throw an exception. So, for our example, please include an empty providers property in the `TimerWidgetComponent` decorator configuration:

app/timer/timer-widget.component.ts

```
@Component({
  selector: 'pomodoro-timer-widget',
  styleUrls: ['app/timer/timer-widget.component.css'],
  providers: [],
  template: `  …  `
})
```

This issue might be addressed by the Angular 2 team in the future, but in the meantime we need to proceed this way.

Tip

Need to override a test component's directives or template?

You can override other elements of the test component instance with the methods `overrideTemplate()`, `overrideView()` (which gives you access to override the literal defining things such as styles), or `overrideDirectives()`. Their signature follows pretty much the same convention, where we define first the component type and then, as a second argument, the replacement we need for the component original value.

Please refer to the official API documentation for further details if required.

Testing routes

Just like components, routes play an important role in the way our applications deliver an efficient user experience. As such, testing routes becomes paramount to ensure a flawless performance. Trying to declare the `Router` token as a provider would turn into an exception, so we basically need to somehow inform our test injector about what should it use as root router and root component, bringing in all the dependencies required by these two types, aside from mocking the `Location` service with a more specialized service which is the `SpyLocation` service. The following example clarifies all this (disregard the `LoginComponent` import for now, as we will use it in the next section):

app/app.component.spec.ts

```
import AppComponent from './app.component';
import { LoginComponent } from './login/login';
import { provide } from '@angular/core';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import {
  Router,
  RouteRegistry,
  ROUTER_PRIMARY_COMPONENT } from '@angular/router-deprecated';
import { Location } from '@angular/common';
import { SpyLocation } from '@angular/common/testing';
import { RootRouter } from '@angular/router-deprecated';

describe('AppComponent', () => {
  let location: Location, router: Router;

  // We override the Router and Location providers and its own
  // dependencies in order to instantiate a fixture router to
  // trigger routing actions and a location spy
  beforeEachProviders(() => [
    RouteRegistry,
    provide(Location, { useClass: SpyLocation }),
    provide(Router, { useClass: RootRouter }),
    provide(ROUTER_PRIMARY_COMPONENT, { useValue: AppComponent })
  ]);

  // We instantiate Router and Location objects before each test
  beforeEach(inject([Router, Location], (_router, _location) => {
    router = _router;
    location = _location;
  }));

  // Specs with assertions
  it('can navigate to the main tasks component', done => {
```

```
      // We navigate to a component and check the resulting
      // state in the URL
      router.navigate(['TasksComponent'])
        .then(() => {
          expect(location.path()).toBe('/tasks');
          done();
        })
        .catch(e => done.fail(e));
  });
});
```

Testing routes by URL

In our previous example, we tested how we could navigate to a named route and check if the router resulting URL matched the expected state. But sometimes we want to check whether we have actually loaded the component we aimed to reach. The following example takes the opposite approach: we navigate to a URL and check the resulting component type. Do you remember we imported the `LoginComponent` token previously? Now, we'll put it to good use in our next test assertion.

Please append the following spec to `app/app.component.spec.ts`:

```
...
it('should be able to navigate to the login component', done => {
  // We navigate to an URL and check the resulting state in the URL
  router.navigateByUrl('/login').then(() => {
    expect(router.currentInstruction.component.componentType)
      .toBe(LoginComponent);
    done();
  }).catch(e => done.fail(e));
});
```

Testing redirections

What if we want to check whether a redirection actually works? No worries, we just need to navigate by URL to the path triggering the redirection and then check either the type of the router current instruction or the resulting location path.

Please append the following spec to `app/app.component.spec.ts`:

```
it('should redirect "/" requests to the tasks component', done => {
  // We navigate to an URL and check the resulting state in the URL
  router.navigateByUrl('/').then(() => {
    expect(location.path()).toBe('/tasks');
    done();
  }).catch(e => done.fail(e));
});
```

Testing services

Services are also a subject of testing in our applications. One of the traits that make services so unique in the Angular-land is that they do not necessarily need to rely on Angular 2 itself. Unless a service needs to take advantage of Angular 2's DI machinery to leverage other Angular modules such as HTTP, it will be pretty much a regular, framework-agnostic JavaScript class.

This makes testing services with no dependencies a breeze, where, just like we did when testing pipes, we need to instantiate the class on every test spec and test its properties and the functionality of its methods.

Let's see all this through an actual example, where we will test the simplest service we have on store:

app/shared/services/settings.service.spec.ts

```
import SettingsService from './settings.service';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';

describe('shared:SettingsService', () => {
  let settingsService: SettingsService;

  beforeEach(() => {
    settingsService = new SettingsService();
  });

  it('should provide the duration for each pomodoro', () => {
    expect(settingsService.timerMinutes).toBeDefined();
    expect(settingsService.timerMinutes).toBeGreaterThan(0);
    expect(settingsService.timerMinutes).not.toBeNaN();
  });

  it('should provide plural mappings for tasks', () => {
    const tasksPluralMappings = settingsService.pluralsMap['tasks'];
    const actualProperties = Object.keys(tasksPluralMappings).sort()
    const expectedProperties = ['=0', '=1', 'other'].sort();

    expect(tasksPluralMappings).toBeDefined();
    expect(actualProperties).toEqual(expectedProperties);
  });
});
```

Here, we are testing whether there is an actual numeric value configured in the `timerMinutes` property and whether the `tasksPluralMappings` contains all the mappings we expect it to

have. We could (and definitely should) conduct more tests, but for the time being it is fine to leave the test like this and then focus on an important detail. If we had wanted to leverage the Angular 2 testing machinery for this test, we could have instantiated the `SettingService` object like this:

```
beforeEachProviders(() => [SettingsService]);

beforeEach(inject([SettingsService],
  (_settingsService: SettingsService) => {
    settingsService = _settingsService;
  }
));
```

This syntax will remind you of what you saw in the previous sections. Ultimately, take the approach that suits your coding style. Obviously, the latter will require less refactoring as our service class evolves and begins to demand injected dependencies, in which case making use of the `beforeEachProviders()` function will become paramount.

Testing asynchronous services

The previous example showcased how we can test the most basic bare-bones service we can come up with, but in reality two of the most common traits of custom services are that they usually rely on other dependencies to provide their functionality. In addition, these functionalities are most of the time based on asynchronous methods that connect to third party services. Regardless of the interface exposed by these asynchronous members (callbacks, emitted events, promises, or observables), testing services like these is not hard at all, as we can see in the following example where we test the different API endpoints of `AuthenticationService`. The code is as follows:

app/shared/services/authentication.service.spec.ts

```
import AuthenticationService from './authentication.service';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';

describe('shared:AuthenticationService', () => {
  let authenticationService: AuthenticationService;

  beforeEachProviders(() => [
    AuthenticationService
  ]);

  beforeEach(inject(
    [AuthenticationService], (_authenticationService) => {
      authenticationService = _authenticationService;
    }
  ));

  it('should reject invalid credentials', done => {
    authenticationService.login({
      username: 'foo',
      password: 'bar'})
      .then(success => {
        expect(success).toBeFalsy();
        done();
      });
  });

  describe('emits an event upon user auth status changes', () => {

    it('that should be truthy for successful logins', done => {
        authenticationService
        .userIsloggedIn
        .subscribe((authStatus: boolean) => {
          expect(authStatus).toBeTruthy();
          done();
```

```
        });

        authenticationService.login({
          username: 'john.doe@mail.com',
          password: 'letmein'
        });
      });

      it('that should be falsy for failed logins', done => {
        authenticationService
        .userIsloggedIn
        .subscribe((authStatus: boolean) => {
          expect(authStatus).toBeFalsy();
          done();
        });

        authenticationService.login({
          username: 'foo',
          password: 'bar'
        });
      });
    });
  });
});
```

The test must seem lengthy, but it is actually quite simple, since we are just putting into practice all that we know by now about unit testing. The only part worth remarking is how we wrap the expectation assertion in the `subscribe()` method of the `userIsLoggedIn` event emitter member, so the assertion will only be evaluated once an event is emitted and shines through the subscription function. The code is as follows:

```
authenticationService
  .userIsloggedIn
  .subscribe((authStatus: boolean) => {
    expect(authStatus).toBeTruthy();
    done();
  });
```

We then conduct an authentication request in the following block, so the `subscribe()` function emits the expected event:

```
authenticationService.login({
  username: 'john.doe@mail.com',
  password: 'letmein'
});
```

Last but not least, please notice how we have nested a `describe()` suite within another `describe()` suite. This is quite common whenever it makes sense to group test specs by area of functionality, easing the task of disabling tests if required.

Mocking Http responses with MockBackend

The previous example is a bit contrived since our `AuthenticationService` module was in fact a mock by itself, with no real implementation whatsoever. In a real scenario, the `AuthenticationService` should implement an asynchronous method sending a POST request to an authentication service. Let's update our current service implementation to include this feature under a different method name, so it does not collide with the current `login()` implementation and its tests. The code is as follows:

app/shared/services/authentication.service.ts

```
...
httpLogin(credentials): Promise<boolean> {
  return new Promise(resolve => {

    const url = '/api/authentication'; // Or your own API Auth url
    const body = JSON.stringify(credentials);
    const headers = new Headers({'Content-Type':'application/json'});
    const options = new RequestOptions({ headers: headers });

    this.http.post(url, body, options)
      .map(response => response.json())
      .subscribe(authResponse => {
          let validCredentials: boolean = false;

          if(authResponse && authResponse.token) {
            validCredentials = true;
            window.sessionStorage.setItem(
              'token',
              authResponse.token
            );
          }

          this.userIsloggedIn.emit(validCredentials);
          resolve(validCredentials);
        },
        error => console.log(error)
      );
  });
}
...
```

In our new implementation of `AuthenticationService`, a new asynchronous method named `httpLogin()` performs an actual HTTP POST request to an auth service of our choice and submits the credentials in JSON format, resolving a promise with a Boolean value after persisting the token in the local session storage.

Tip

For the sake of reusability, the method should just resolve to the HTTP response once fetched and it will be up to the method's clients to decide how to persist the information contained in

the response and what to do next. For the sake of brevity, let's leave our method like this.

With our new shiny asynchronous method, there are some new challenges:

- First, we need to declare the dependencies required for allowing HTTP connections
- Second, our new method performs an actual HTTP request to a remote service which is out of the scope of our testing capabilities, so we need to be able to intercept such request and return a customized mocked response to fulfil our tests

Regarding the former, we already saw in previous sections how to declare providers and instantiate an Http dependency through the injector using MockBackend in the dependency constructor.

However, it is time to harness all the functionality that MockBackend can provide for intercepting HTTP connections and mock responses of our own. Let's get back to authentication.service.spec.ts and replace the current implementation of beforeEachProviders() and beforeEach() after importing some more tokens you're already familiar with:

app/shared/services/authentication.service.spec.ts (updated)

```
import AuthenticationService from './authentication.service';
import { provide } from '@angular/core';
import {
  describe,
  expect,
  it,
  inject,
  beforeEach,
  beforeEachProviders } from '@angular/core/testing';
import {
  Http,
  BaseRequestOptions,
  Response,
  ResponseOptions }  from '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';
import 'rxjs/add/operator/map';

describe('shared:AuthenticationService', () => {
  let authenticationService: AuthenticationService;
  let mockBackend: MockBackend;

  beforeEachProviders(() => [
    MockBackend,
    BaseRequestOptions,
    provide(Http, {
      useFactory: (
        backend: MockBackend,
        options: BaseRequestOptions
      ) => {
        return new Http(backend, options);
      },
```

```
      deps: [MockBackend, BaseRequestOptions]
    }),
    AuthenticationService
  ]);

  beforeEach(inject(
    [MockBackend, AuthenticationService],
    (_mockBackend, _authenticationService) => {
      authenticationService = _authenticationService;
      mockBackend = _mockBackend;
    }
  ));

  it('can fetch a valid token when querying the Auth API', done => {
    const mockedResponse = new ResponseOptions({
      body: '{"token": "eyJhbGciOi"}'
    });

    mockBackend.connections.subscribe(
      (connection: MockConnection) => {
        if(connection.request.url === '/api/authentication') {
          connection.mockRespond(new Response(mockedResponse));
        }
      }
    );

    authenticationService.httpLogin({
      username: 'foo',
      password: 'bar'
    }).then(success => {
        expect(success).toBeTruthy();
        done();
      },
      error => done.fail(error)
    );
  });

  // Rest of test specs remain the same below
  // ...
```

First, we import everything that is required to interact with HTTP-based services. The
`beforeEachProviders()` and the `beforeEach()` implementations have no difference with what
we already saw when overviewing `timer-widget.component.test.ts`. Perhaps the only
nuance worth remarking is the fact that we bind a new instance of `MockBackend` to the
`mockBackend` variable on every test execution. It is required because we will be using it in the
newly introduced test spec. Let's review it in more detail. First, we define a mocked response
with some fake data. We will be using this mock response later on when performing actual
HTTP requests:

```
const mockedResponse = new ResponseOptions({
  body: '{"token": "eyJhbGciOi"}'
});
```

The `MockBackend` objects expose a connections property of type `EventEmitter`, which emits a

`MockConnection` event object every time it detects an attempt to perform a XHR connection through `Http` (which, as we know now, is using `mockBackend` to perform backend connections). This `MockConnection` object contains relevant information about the request attempt that we can use to refine our test and return a specific response tailored to the necessities of our testing scenario. To do so, we will use the `mockRespond()` of the `connection` object itself. The code is as follows:

```
mockBackend.connections.subscribe(
  (connection: MockConnection) => {
    if(connection.request.url === '/api/authentication') {
      connection.mockRespond(new Response(mockedResponse));
    }
  }
);
```

With all these elements in place, performing an actual request with our service methods and evaluating the responses becomes an easy task:

```
authenticationService.httpLogin({
  username: 'foo',
  password: 'bar'
}).then(success => {
  expect(success).toBeTruthy();
  done();
},
  error => done.fail(error)
);
```

Testing directives

The last leg of our journey into the world of unit testing Angular 2 elements will cover directives. Directives will be usually quite straightforward in their overall shape, being pretty much 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.

We could create a stub component for the purpose of the test and then bind the directive on it, either directly upon defining it or by leveraging the `overrideDirectives()` of the `TestComponentBuilder` instance object we will compose for the test. In that sense, the component, as the host element for the directive, will proxy our test operations, so we will not delve deeper into this approach after reviewing component testing in the previous sections.

Another approach is to leverage the host bindings and listeners our directive takes action on, and test the bound methods to these decorators to see if they provide the functionality required. Let's look at an actual example of this approach by testing the `TaskTooltipDirective` module:

app/tasks/task-tooltip.directive.spec.ts

```
import { Task } from '../shared/shared';
import TaskTooltipDirective from './task-tooltip.directive';
import {
  describe,
  expect,
  it,
  beforeEach } from '@angular/core/testing';

  describe('shared:TaskTooltipDirective', () => {
    let taskTooltipDirective: TaskTooltipDirective;

    beforeEach(() => {
      taskTooltipDirective = new TaskTooltipDirective();
    });

    it('should update a given tooltip upon mouseover', done => {
      let mockTooltip = { innerText: '' };
      taskTooltipDirective.task = <Task>{ name: 'Foo' };
      taskTooltipDirective.taskTooltip = mockTooltip;

      taskTooltipDirective.onMouseOver();
      expect(mockTooltip.innerText).toBe('Foo');

      done();
    });

    it('should restore a given tooltip upon mouseout', done => {
      let mockTooltip = { innerText: 'Foo' };
      taskTooltipDirective.task = <Task>{ name: 'Bar' };
      taskTooltipDirective.taskTooltip = mockTooltip;
```

```
      taskTooltipDirective.onMouseOver();
      expect(mockTooltip.innerText).toBe('Bar');

      taskTooltipDirective.onMouseOut();
      expect(mockTooltip.innerText).toBe('Foo');

      done();
    });
  });
```

Here, we instantiate the directive object directly, since it has no dependencies that require us to use the injector instead. Since all the operations performed by this directive are governed by the class methods decorated with `@HostListener()` decorators, we just need to feed the directive class input members with mock data and see if we obtain the desired behavior.

The road ahead

This last test example wraps up our journey into unit testing with Angular 2, but keep in mind that we have barely scratched the surface. Testing web applications in general and Angular 2 applications in particular poses a myriad of scenarios that need a specific approach most of the times. Remember that if a specific test requires a cumbersome and convoluted solution, we are probably facing a good case for a module redesign instead.

Where should we go from here? There are several paths to compound our knowledge of web application testing in Angular 2 and become great testing ninjas.

Using Jasmine in combination with Karma

So far, we have used the Jasmine HTML spec runner to execute our tests and get a results report. While this is perfectly fine for smaller projects, the HTML spec runner might not be the best solution for bigger projects, especially if we want our tests to be re-executed automatically when code changes or we need to hook up our tests layer with a continuous integration server.

At this point you will want to use a more powerful and faster spec runner without compromising your project. For that reason, your best bet might be picking up Karma as a spec runner. Used by the Angular team itself, it plays well with Jasmine and other testing frameworks such as Mocha or QUnit. It also features a simple but powerful configuration setup with support for automatic spec scanning, file watching, multiple report outputs, and advanced extensibility with plugins.

For those in need to hook up their application with continuous integration servers, Karma also provides adapters for Jenkins, Travis, or Semaphore.

You can find further information at https://karma-runner.github.io.

Introducing code coverage reports in your test stack

How can we know how far do our tests go on testing the application? Are we sure we are not leaving any piece of code untested and if so, is it relevant? How can we detect the pieces of code that fall outside the scope of our current tests so we can better assess if they are worth testing or not?

These concerns can be easily addressed by introducing code coverage reporting in our application tests stack. A code coverage tool aims to track down the scope of our unit testing layer and produce an educated report informing of the overall reach of your test specs and what pieces of code still remain uncovered.

There are several tools for implementing code coverage analysis in our applications, Blanket (http://blanketjs.org), and Istanbul (https://gotwarlost.github.io/istanbul) the most popular ones at this time. In both cases, the installation process is pretty quick and easy.

Implementing E2E tests

In this chapter, we saw how we could test certain parts of the UI by evaluating the state of the DOM. This gives us a good idea of how things would look like from the end user's point of view, but ultimately this is just an uneducated guess.

End-to-end (E2E) testing is a methodology for testing web applications using an automated agent that will programmatically follow the end user's flow from start to finish. Contrary to what unit testing poses, the nuances of the code implementation are not relevant here, since E2E testing entails testing our application from start to finish from the user's endpoint . This approach allows us to test the application in an integrated way. While unit testing focuses on the reliability of each particular piece of the puzzle, E2E testing does assess the integrity of the puzzle as a whole, finding integration issues between components that are frequently overlooked by unit tests.

The Angular team built for the previous incarnation of the Angular framework a powerful tool named Protractor (https://docs.angularjs.org/guide/e2e-testing), which is defined as follows:

> *"..an end to end test runner which simulates user interactions that will help you verify the health of your Angular application."*

The tests syntax will become pretty familiar since it also uses Jasmine for putting together test specs. Unfortunately, E2E sits outside the scope of this book, but there are several resources you can rely on to expand your knowledge on the subject. In that sense, we recommend the book *Angular 2 Test-driven development, Packt Publishing,* which provides broad insights on the use of Protractor to create E2E test suites for our Angular 2 applications.

Summary

We are at the end of our journey, and it's been a long but exciting one without any shade of doubt. In this chapter, you saw the importance of introducing unit testing in our Angular 2 applications, the basic shape of a unit test, and the process of setting up Jasmine for our tests. You also saw how to code powerful tests for our components, directives, pipes, routes, and services. We also discussed new challenges in your path for mastering Angular 2. It is fair to say that there is still a long road ahead, and it is definitely an exiting one.

The end of this chapter is also the end of this book, but the experience continues beyond its boundaries. Angular 2 is still a pretty young framework and as such all the great things that it will bring to the community are yet to be created. Hopefully, you will be one of those creators. If so, please let the author know.

Thanks for taking the time for reading this book.

Index

A

- Angular 2
  - defining /
  - URL /
  - TypeScript classes /
  - metadata decorators, defining /
  - TypeScript, compiling into browser-friendly JavaScript /
  - HTML container /
  - examples /
  - template, editing /
  - directives /
  - dependency injection, defining /
  - dependencies, injecting across component tree /
  - providers, overriding in injectors hierarchy /
  - injector support, extending to custom entities /
  - applications, initializing with bootstrap() /
  - references /
  - matcher functions, defining /
- Angular 2 cheat sheet
  - URL /
- Angular 2 components
  - defining /
  - productivity, improving /
  - component methods /
  - data updates /
  - interactivity, adding /
  - data output, improving in view /
- Angular 2 router bundle
  - implementing /
- AnimationBuilder
  - components, animating with /
  - CssAnimationBuilder API /
  - animation state, tracking with Animation class /
- animations
  - creating, with plain vanilla CSS /

B

- Blanket
  - URL / [Introducing code coverage reports in your test stack](#)
- Bootstrap / [Installing dependencies](#)
- bootstrap method
  - actions, defining / [Putting everything together](#)

C

D

E

F

G

H

I

J

- Jasmine
  - URL / [Dependency injection in unit tests](#)
  - using, with Karma / [Using Jasmine in combination with Karma](#)
- John Papa
  - URL / [Introducing the Pomodoro App directory structure](#)
- JSBIN
  - URL / [Observables in a nutshell](#)
- Json pipe / [The JSON pipe](#)

K

- Karma
  - URL /

L

N

O

P

R

S

T

U

V

W