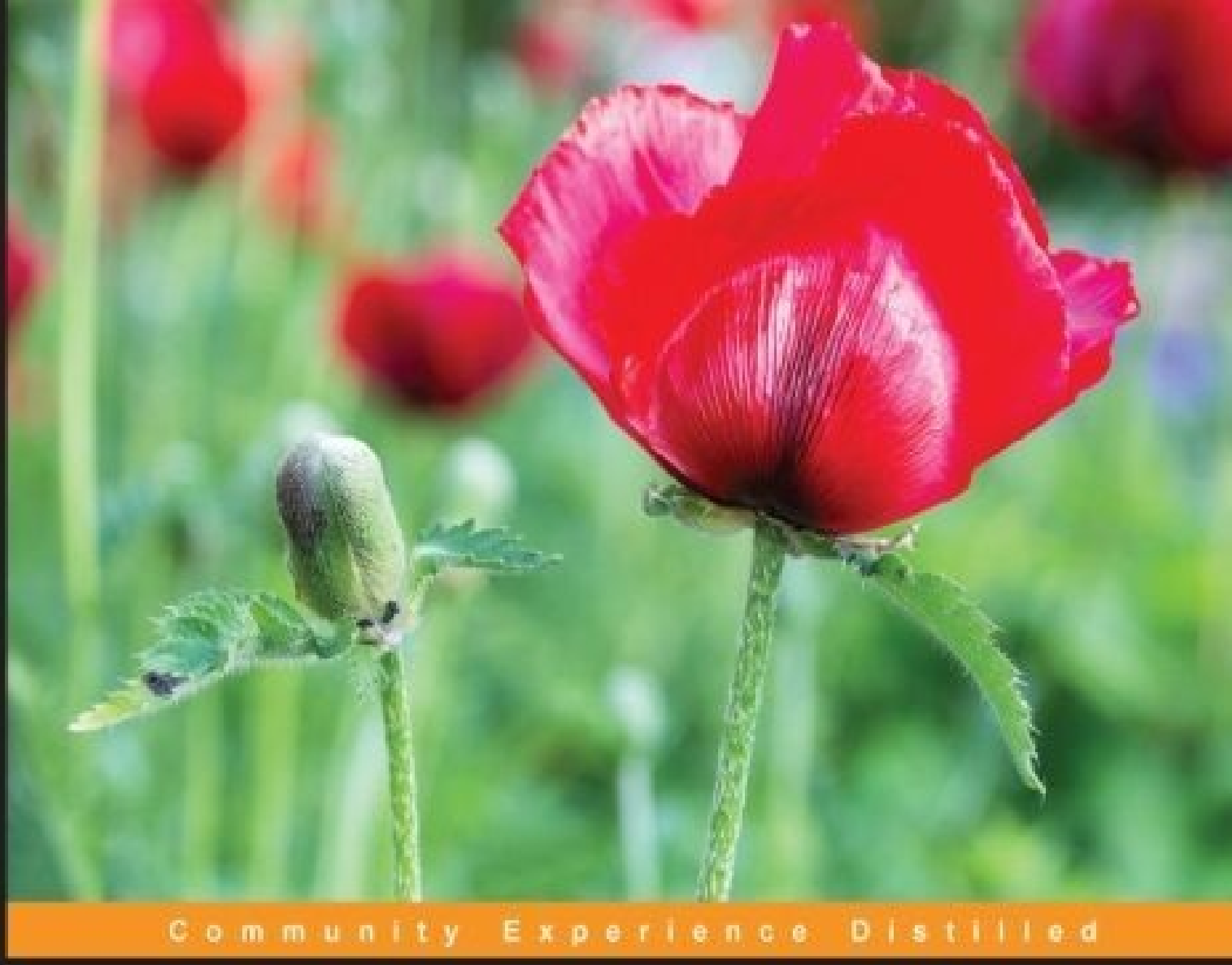


Community Experience Distilled

Mastering Angular 2 Components

Learn to build component-based user interfaces of the future using Angular 2

This book is based on RC1

oddEVEN

Gion Kunz

[PACKT] open source
PUBLISHING

Mastering Angular 2 Components

Table of Contents

Mastering Angular 2 Components

Mastering Angular 2 Components

First published: June 2016

Production reference: 1280616

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-464-1

www.packtpub.com

Credits

Author

Gion Kunz

Reviewer

Carlos Morales

Commissioning Editor

Sarah Crofton

Acquisition Editors

Aaron Lazar

Larissa Pinto

Content Development Editor

Samantha Gonsalves

Technical Editor

Madhunikita Sunil Chindarkar

Copy Editor

Priyanka Ravi

Project Coordinator

Sanchita Mandal

Proofreader

Safis Editing

Indexer

Monica Ajmera Mehta

Graphics

Disha Haria

Production Coordinator

Melwyn Dsa

Cover Work

Melwyn Dsa

About the Author

Gion Kunz has years of experience with web technologies and is totally in love with web standards. With over 10 years of experience of writing interactive user interfaces using JavaScript, he constantly evaluates new approaches and frameworks. He's worked with AngularJS for over 3 years now and is one of the earliest adopters of Angular 2. Gion speaks about Angular 2 at conferences, and he helps with the organization of the Zurich Angular Meetup group in Switzerland.

He currently works for the start-up company oddEVEN in Zurich, where they help customers build websites and applications. Besides working for oddEVEN, Gion is a head instructor at the SAE Institute in Zurich and loves to get his students enthusiastic about the Web.

He is also the creator of the responsive charting library Chartist, and he loves to contribute to the open source community whenever he finds time.

When Gion is not busy with web technologies, you can probably find him at his home music studio, outdoors, fishing, or spending quality time with his girlfriend and their cute little dog.

I would like to thank my girlfriend, Nathalie, for her ongoing support and her patience with all my efforts spent on this book. I really appreciate all that you did for both of us during this time and that you compensated the energy loss I've brought into our relation.

About the Reviewer

Carlos Morales started programming in BASIC when he was 6 years old with a Sinclair ZX Spectrum+ that he still owns. He's loved the technology ever since. Professionally, he has worked for more than 15 years in different roles, always around web applications. He fell in love with Angular, and he founded the Angular Meetup group in Zürich, which is one of the most attended Meetups in the city.

Soy ingeniero informático gracias a mi dedicación, pero sobretodo a mis padres.

www.PacktPub.com

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

Preface

Web components have long been touted as the next great leap forward in web development. With Angular 2, we're closer than ever. Over the past couple of years, there's been a lot of buzz around web components for quite some time in the web development community. New component-style directives in Angular 2 will change developers' workflows and their thinking about shared and reusable blocks of custom HTML in the shadow DOM. Ours is the first book that guides developers along this path. It's also a practical way to learn, giving readers the chance to build components of their own. With *Mastering Angular 2 Components*, learners will get ahead of the curve in a new wave of web development by tightly focusing on an area that's the key to unlocking the powers of Angular development.

Mastering Angular 2 Components teaches readers to think componentially. This rich guide to the new component-centric way of doing things in Angular teaches readers how to invent, build, and manage shared and reusable components for their web projects. This book will change how developers think about how to accomplish things in Angular 2, and the reader will be working on useful and fun example components throughout.

What this book covers

Chapter 1, *Component-Based User Interfaces*, looks at a bit at the UI development history and provides a brief introduction to component-based user interfaces in general. We will see how Angular 2 handles this concept.

Chapter 2, *Ready, Set, Go!*, will get the reader started on their journey toward building an Angular 2 component-based application. It covers the basic elements of structuring an application with components.

Chapter 3, *Composing with Components*, is where the reader will start to structure the user interface into its basic pieces. The reader will then go on to compose an application using components by organizing an application layout into components, establishing the composition of components using QueryList, and creating a reusable tab component to structure the application interface.

Chapter 4, *No Comments, Please!*, is where the reader will learn how to build a commenting system using components. They will learn to create a component to list comments and also to create new comments.

Chapter 5, *Component-Based Routing*, explains how components react to routing and will enable the reader to add simple routing to the existing components in the task management application. The reader will also work on the login process and build an understanding to protect components using the router.

Chapter 6, *Keeping Up with Activities*, covers the creation of components that will visualize activity streams on project and task level.

Chapter 7, *Components for User Experience*, is where the reader will create many small reusable components that will have a great effect on the overall user experience of the task management application. This includes in-place editing of text fields, infinite scroll, popup notification, and drag and drop support.

Chapter 8, *Time Will Tell*, focuses on creating time-tracking components that helps estimate time on a project and task level but also for users to log the time spent on tasks.

Chapter 9, *Spaceship Dashboard*, focuses on creating components to visualize some data in the task management application using the third-party library Chartist.

Chapter 10, *Making Things Pluggable*, is where the reader will learn about an approach to make components pluggable using a simple but powerful pattern. With a DIY plugin architecture for Angular 2 components, we make our task management system extensible.

Chapter 11, *Putting Things to the Test*, covers some basic approaches to testing Angular 2 components. We will see the options for mocking/overriding specific parts of a component

for testing.

Appendix, *Task Management Application Source Code*, contains all the information you'll need to download and install the source code that comes with this book. You'll also find instructions to use and troubleshoot the code in there.

What you need for this book

This book will need a basic installation of Node.js on your Windows, Mac, or Linux machine.

Who this book is for

This book is for Angular developers who already have a good understanding of basic frontend web technologies, such as JavaScript, HTML, and CSS. You will learn about the new component-based architecture in Angular 2 and how to use it to build modern and clean user interfaces.

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: "You have a `Fisher` class and a `Developer` class, both of which hold specific behaviors."

A block of code is set as follows:

```
class Fruit {
  constructor(name) { this.name = name; }
}
const apple = new Fruit('Apple');
```

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>
<template id="template">
  <h1>This is a template!</h1>
</template>
</body>
```

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

```
npm install jspm --save-dev
jspm init
```

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: "If everything goes well, you will have an open web browser that shows **Hello World!**."

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 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

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Mastering-Angular-2-Components. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/MasteringAngular2Components_Colo

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.

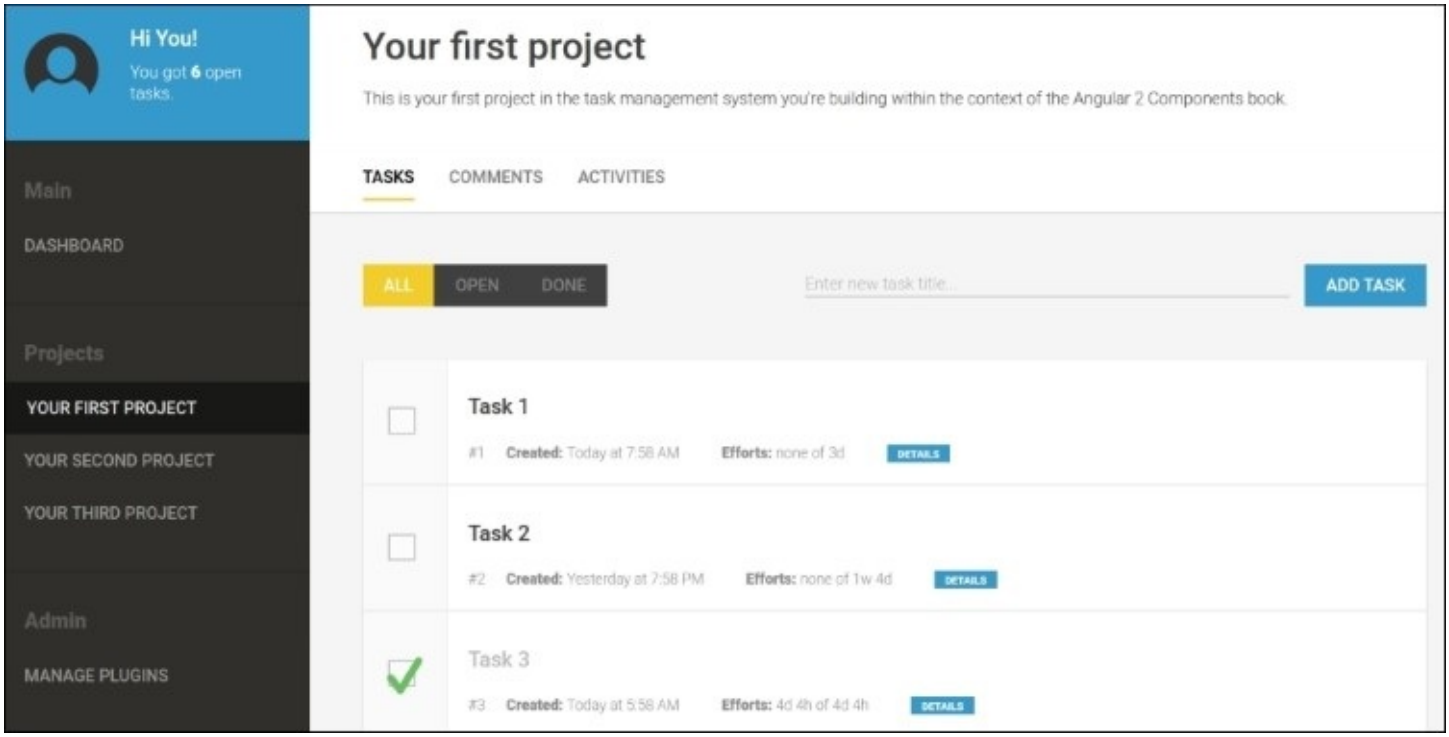
Questions

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. Component-Based User Interfaces

Although we'll cover a lot of Angular-related topics in this book, the focus will be mainly on creating component-based user interfaces. It's one thing to understand a framework, such as Angular 2, but it's a whole different thing to establish an effective workflow using a component-based architecture. In this book, I'll try to explain the core concepts behind Angular 2 components and how we can leverage this architecture to create modern, efficient, and maintainable user interfaces.

Besides learning all the necessary concepts behind Angular 2, we will together create a task-management application from scratch. This will allow us to explore different approaches to solve common UI problems using the component system that is provided by Angular 2.



A preview of the task management application that we are going to build

In this chapter, we will take a look at how component-based user interfaces help us build greater applications. Over the course of this book, we will build an Angular 2 application together, where we will use the component-based approach to its full potential. This chapter will also introduce you to the technologies that are used in this book. The topics that we will cover in this chapter are as follows:

- An introduction to component-based user interfaces
- Encapsulation and composition using component-based user interfaces
- Evolution of UI frameworks
- The standard and Web components

- An introduction to the Angular 2 component system
- Writing your first Angular 2 component
- An overview and history of ECMAScript and TypeScript
- ECMAScript 7 decorators as meta annotations
- An introduction to Node.js-based tooling using JSPM and SystemJS

Thinking of organisms

Today's user interfaces do not consist of just a bunch of form elements that are cobbled together onto a screen. Modern users experience design and innovative visual presentations of interactive content challenges technology more than ever.

Sadly, we almost always tend to think in pages when we flesh out concepts for web applications, such as the pages within a printed book. Well, this is probably the most efficient way to convey information for this kind of content and medium. You can skim through the pages one by one without any real physical effort, read paragraph by paragraph, and just scan through the chapters that you don't find interesting.

The problem with thinking in pages too much is that this concept, which is borrowed from books, does not really translate well to how things work in the real world. The world is created from organisms that form a system of organisms together. This system itself forms an organism again, just on a higher level.

Take our bodies as an example. We mostly consist of independent organs that interact with each other using electrical and chemical signals. Organs themselves consist of proteins that on their own work like a machine to form a system. Down to the molecules, atoms, protons, and quarks, we can't really tell where one starts and where it ends. What we can tell for sure is that it's all about systems of organisms with interdependencies, and it is not about pages.

I like to view user interfaces as systems of organisms. Whether if, where, and how they are distributed to pages is subordinate while designing them. Also, they should work independently, and they should interact with each other on an interdependent level.

Components – The organs of user interfaces

"We're not designing pages, we're designing systems of components."

--Stephen Hay

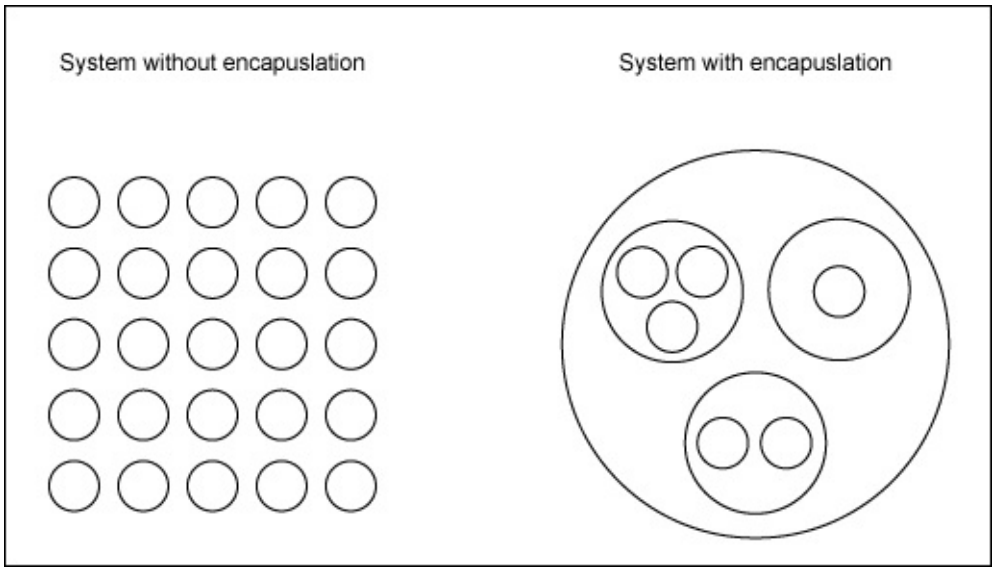
This quote by Stephen Hay from BDConf in Orlando 2012 brings it to the point. Interface design is really not about pages. To create efficient user interfaces for not only the users but also the developers who maintain them, we need to think in systems of components. Components are independent, but they can interact with each other and create larger components when they are arranged together. We need to look at user interfaces holistically and using components enables us to do this.

In the following topics, we're going to explore a few fundamental aspects of components. Some of these are already known from other concepts, such as **object-oriented programming** (**OOP**), but they appear in a slightly different light when thinking about components.

Encapsulation

Encapsulation is a very important factor when thinking about maintenance in a system. Having a classical OOP background, I've learned that encapsulation means bundling logic and data together into an isolated container. This way, we can operate on the container from the outside and treat it like a closed system.

There are many positive aspects of this approach when it comes to maintainability and accessibility. Dealing with closed systems is important for the organization of our code. However, this is even more importantly because we can organize ourselves while working with code.



I have a pretty bad memory, and it's very important for me to find the right focus level when working on code. Immediate memory research told us that the human brain can remember about seven items at once on an average. Therefore, it's crucial for us to write code in such a way that it allows us to focus on fewer and smaller pieces at once.

A clear encapsulation helps us in organizing our code. We can maybe forget all the internals of the closed system and about the kind of logic and data that we've put into it. We can focus only on its surface, which allows us to work on a higher-abstraction level. Similar to the previous figure, without using a hierarchy of encapsulated components, we'd have all our code cobbled together on the same level.

Encapsulation encourages us to isolate small and concise components and build a system of components. During development, we can focus on the internals of one component and only deal with the interface of other components.

Sometimes, we forget that all the organization of the coding we actually perform is for ourselves and not for the computer that runs this code. If this was for the computer, then we

would probably all start writing in machine language again. A strong encapsulation helps us access specific code easily, focus on one layer of the code, and trust the underlying implementations within capsules.

The following JavaScript example shows you how to use encapsulation to write maintainable applications. Let's assume that we are in a T-shirt factory, and we need some code to produce T-shirts with a background and foreground color. This example uses some new language features of ECMAScript 6. If you're not familiar with the language features of ECMAScript 6, don't worry too much at this point. We will learn about these later in this chapter:

```
// This class implements data and logic to represent a colour
// which establishes clean encapsulation.
class Colour {
  constructor(red, green, blue) {
    Object.assign(this, {red, green, blue});
  }

  // Using this function we can convert the internal colour values
  // to a hex colour string like #ff0000 (red).
  getHex() {
    return '#' + Colour.getHexValue(this.red) + Colour.getHexValue(this.green) +
      Colour.getHexValue(this.blue);
  }

  // Static function on Colour class to convert a number from
  // 0 to 255 to a hexadecimal representation 00 to ff
  static getHexValue(number) {
    const hex = number.toString(16);
    return hex.length === 2 ? hex : '0' + hex;
  }
}

// Our TShirt class expects two colours to be passed during
// construction that will be used to render some HTML
class TShirt {
  constructor(backgroundColour, foregroundColour) {
    Object.assign(this, {backgroundColour, foregroundColour});
  }

  // Function that returns some markup which represents our
  // T-Shirts
  getHtml() {
    return `
      <t-shirt style="background-color: ${this.backgroundColour.getHex()}">
        <t-shirt-text style="color: ${this.foregroundColour.getHex()}">
          Awesome Shirt!
        </t-shirt-text>
      </t-shirt>
    `;
  }
}

// Instantiate a blue colour
const blue = new Colour(0, 0, 255);
```

```
// Instantiate a red colour
const red = new Colour(255, 0, 0);
// Create a new shirt using the above colours
const awesomeShirt = new TShirt(blue, red);
// Adding the generated markup of our shirt to our document
document.body.innerHTML = awesomeShirt.getHtml();
```

Using a clean encapsulation, we can now work with the abstraction of color in our T-shirt. We don't need to worry about how to calculate the hexadecimal representation of colors at the T-shirt level because this is already done by the `Colour` class. This makes your application maintainable and keeps it very open for change.

I really recommend that you read about the SOLID principles if you haven't done so already. As the name already suggests, this assembly of principles is a solid power tool that can change the way you organize code tremendously. You can learn more about the SOLID principles in the book, *Agile Principles, Patterns, and Practices*, by Robert C. Martin.

Composability

Composition is a special kind of reusability. You don't extend an existing component, but you create a new larger component by composing many smaller components together into a system of components.

In OOP languages, composition is often used to get around the multiple inheritance issues that most OOP languages have. Subclass polymorphism is always great until you reach the point where your design does not match the latest requirements in your project. Let's look at a simple example that illustrates this problem.

You have a `Fisher` class and a `Developer` class, both of which hold specific behaviors. Now, you'd want to create a `FishingDeveloper` class that inherits both from `Fisher` and `Developer`. Unless you're using a language that supports multiple inheritance (such as C++ does to a certain extent), you will not be able to reuse this functionality using inheritance. There is no way to tell the language that your new class should inherit from both super classes. Using composition, you can easily solve this problem. Instead of using inheritance, you're composing a new `FishingDeveloper` class that delegates all behavior to an internal `Developer` and `Fisher` instance:

```
class Developer {
  code() {
    console.log(`${this.name} writes some code!`);
  }
}

class Fisher {
  fish() {
    console.log(`${this.name} catches a big fish!`);
  }
}

class FishingDeveloper {
  constructor(name) {
    this.name = name;
    this.developerStuff = new Developer();
    this.fisherStuff = new Fisher();
  }

  code() {
    this.developerStuff.code.bind(this)();
  }

  fish() {
    this.fisherStuff.fish.bind(this)();
  }
}

var bob = new FishingDeveloper('Bob');
bob.code();
bob.fish();
```

Experience has taught us that composition is probably the most efficient way to reuse code. In contrast to inheritance, decoration, and other approaches to gain reusability, composition is probably the least intrusive and the most flexible.

Recent versions of some languages also support a pattern called traits, that is, `mixins`. Traits allow you to reuse certain functionality and attributes from other classes in a way that is similar to multiple inheritance.

If we think about the concept of composition, it's nothing more than designing organisms. We have the two `Developer` and `Fisher` organisms, and we unify their behaviors into a single `FishingDeveloper` organism.

Components, invented by nature

Components, embracing encapsulation, and composition are an effective way to build maintainable applications. Composed from components, applications are very resistant to the negative implications of change, and change is a necessary thing that will happen to every application. It's only a matter of time until your design will be challenged by the effects of change; therefore, it's very important to write code that can handle change as smoothly as possible.

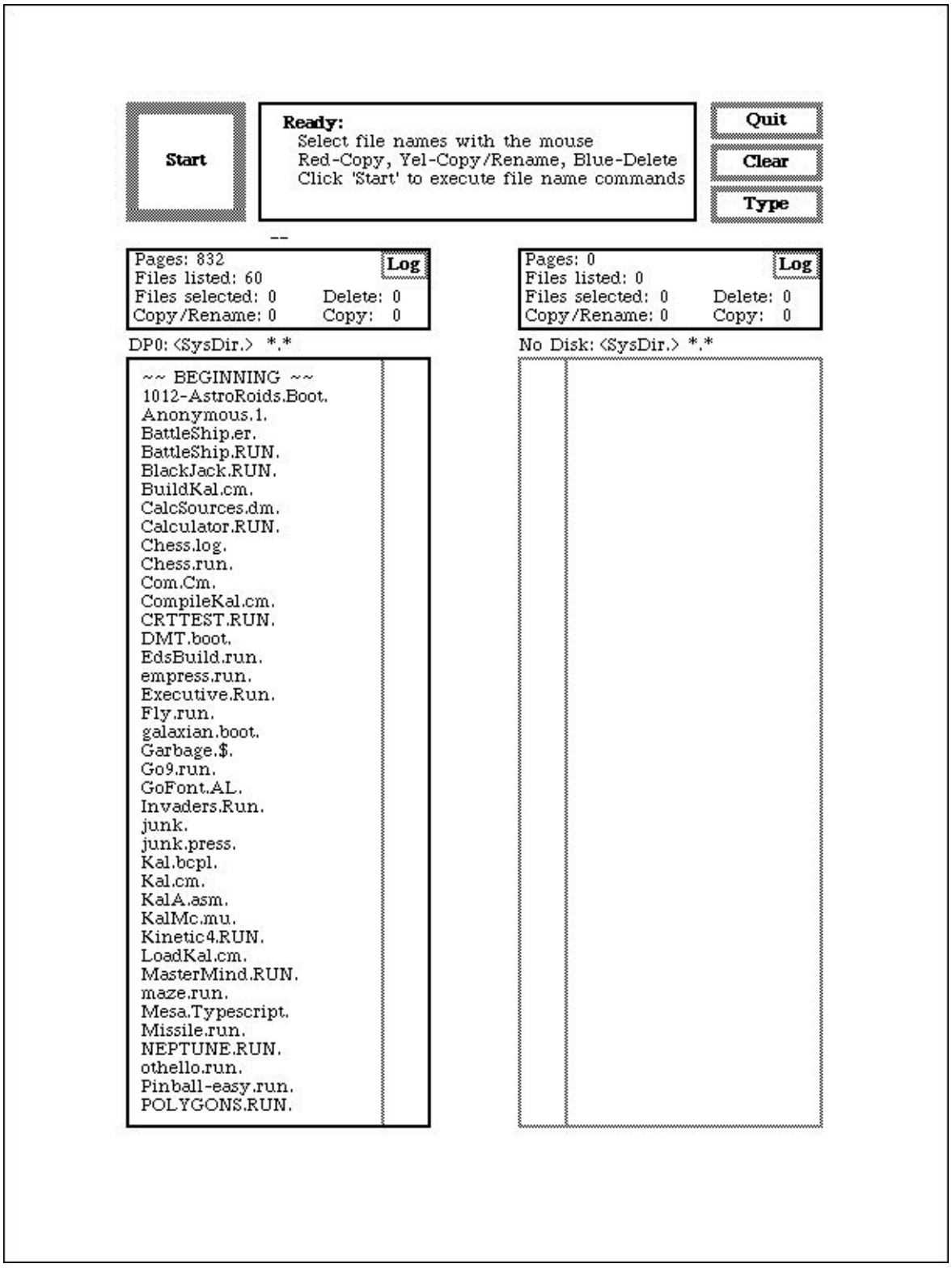
Nature is the best teacher. Almost all the achievements in technological developments have their origin in observations of how nature solves problems. If we look at evolution, it's an ongoing redesign of matter by adapting to outer forces and constraints. Nature solves this by constant change using mutation and natural selection.

If we project the concept of evolution onto developing an application, we can say that nature does actually refactor its code in every single moment. This is actually the dream of every product manager—an application that can undergo constant change but does not lose any of its efficiency.

I believe that there are two key concepts that play a major role in nature that allows it to apply constant change in its design without losing much efficiency. This uses encapsulation and composition. Coming back to the example of our bodies, we can actually tell that our organs use a very clear encapsulation. They use membranes to create isolation, veins to transport nutrition, and synapses to send messages. Also, they have interdependencies, and they communicate with electrical and chemical messages. Most obviously, they form larger systems, which is the core concept of composition.

Of course, there are many other factors, and I'm not a professor in biology. However, I think it's a fascinating thing to see that we have learned to organize our code very similarly to how nature organizes matter.

The idea of creating reusable UI components is quite old, and it was implemented in various languages and frameworks. One of the earliest systems that used UI components was probably the Xerox Alto system back in 1970s. It used reusable UI components that allowed developers to create an application by composing them on a screen where users could interact with them.



The user interface of file manager on the Xerox Alto system from the 1970s.

Early frontend UI frameworks, such as DHTMLX, Ext JS, or jQuery UI implemented components in a more limited fashion that didn't provide great flexibility or extensibility. Most of these frameworks just provided widget libraries. The problem with UI widgets is that

they mostly don't embrace the pattern of composition enough. You can arrange widgets on a page and they provide encapsulation, but with most toolkits, you can't create larger components by nesting them inside each other. Some toolkits solve this by providing a special kind of widget which was mostly called a container. However, this is not the same as a full-fledged component tree that allows you to create systems within systems. Containers were actually meant to provide a visual layout container rather than a composite container to form a larger system.

Usually when working with widgets on a page of our application, we'd have a large controller that controls all these widgets, user input, and states. However, we are left with two levels of composition, and there's no way that we can structure our code more granularly. There is the page and there are the widgets. Having a bunch of UI widgets is simply not enough, and we are almost back to the state where we create pages plastered with form elements.

I've been a user of JavaServer Faces for years, and besides all its problems, the concept of having reusable custom elements was groundbreaking. Using XHTML, one could write so-called composite components that consisted of other composite components or native HTML elements. A developer could gain a fantastic level of reusability using composition. The big issue in my view with this technology was that it did not address the concerns in the frontend enough to become really usable for complex user interactions. In fact, a framework like this should live completely within the frontend.

My UI framework wishlist

Usually when UI frameworks get compared, they get measured against each other based on metrics, such as widget count, theming capabilities, and asynchronous data retrieval features. Every framework has its strengths and weaknesses, but leaving all the extra features aside and reducing it to the core concerns of a UI framework, I only have a few metrics left that I'd like to be assessed. These metrics are, of course, not the only ones that are important in today's UI development, but they also are the main factors toward building a clean architecture that supports the principle of change:

- I can create encapsulated components with clear interfaces
- I can create larger components by composition
- I can make components interact with each other within their hierarchy

If you're looking for a framework which enables you to take full advantage of component-based UI development, you should look for these three key measures.

First of all, I think it's very important to understand the main purpose of the web and how it evolved. If we think of the web in its early days in the 1990s, it was probably only about hypertext. There were very basic semantics that could be used to structure information and display them to a user. HTML was created to hold structure and information. The need for custom visual presentation of information led to the development of CSS right after HTML started being widely used.

It was in the mid 1990s when Brendan Eich invented JavaScript, and it was first implemented in Netscape Navigator. By providing a way to implement behavior and state, JavaScript was the last missing piece for a full web customization:

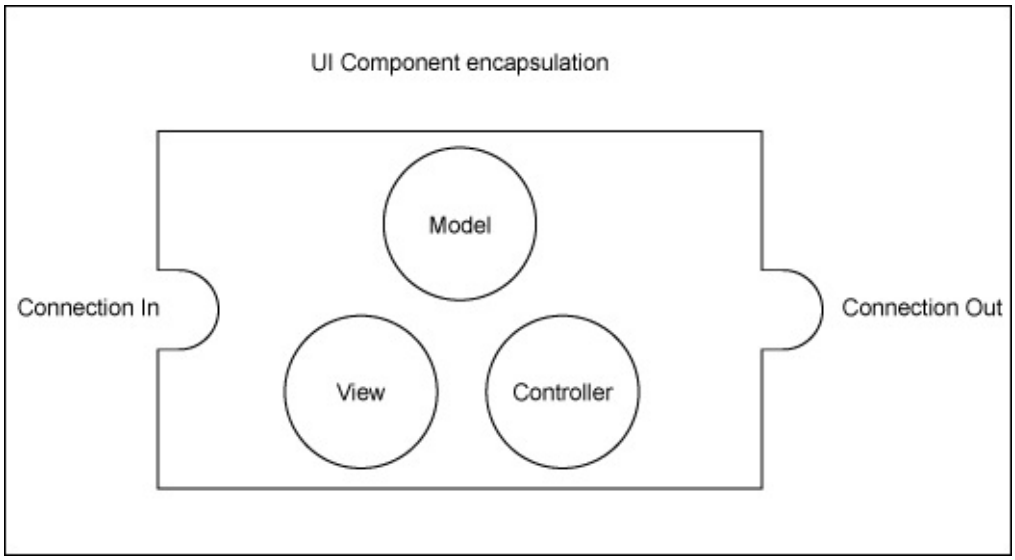
Technology	**Concern**
HTML	Structure and information
CSS	Presentation
JavaScript	Behavior and state

We have learned to keep these concerns as separate as possible in order to maintain a clean architecture. Although there are different opinions on this and some recent technologies also move away from this principle, I believe that a clean separation of these concerns is very important to create a maintainable application.

Leaving this view aside, the standard definition of encapsulation from OOP is just concerned about coupling and isolation of logic and data. This probably applies well to classic software components. However, as soon as we consider a user interface as part of an architecture, there is a new dimension that is added.

Classical MVC frameworks are view centric, and developers organize their code based on pages. You'll probably go ahead and create a new view that represents a page. Of course, your view needs a controller and model, so you'll also create them. The problem with organization by pages is that there's little to no gain of reusability. Once you've created a page and you'd like to reuse only some parts of the page, you will need a way to encapsulate only a specific part of this model—the view and the controller.

UI components solve this problem nicely. I like to see them as a modular approach to MVC. Although they still embrace the MVC pattern, they also establish encapsulation and composability. This way a view is a component itself, but it also consists of components. By composing views of components, one can gain a maximum amount of reusability:



UI components embrace MVC, but they also support encapsulation and composition on a much lower level

Technically, there are some challenges when implementing components with web technologies. JavaScript was always flexible enough to implement different patterns and paradigms. Working with encapsulation and composition isn't an issue at all, and the controlling part and the model of components can easily be implemented. Approaches, such as the revealing module pattern, namespaces, prototypes, or the recent ECMAScript 6 modules, provide all the tools that are needed from the JavaScript side.

However, for the view part of our components, we face some limitations. Although HTML supports great flexibility in terms of composability because the DOM tree is nothing else than a big composition, we have no way to reuse these compositions. We can only create one large composition, which is the page itself. HTML being only the final view that was delivered from the server, this was never really a real concern. Today's applications are much more demanding, and we need to have a fully-encapsulated component running in the browser, which also consists of a partial view.

We face the same problem with CSS. There is no real modularization and encapsulation while writing CSS, and we need to use namespaces and prefixes in order to segregate our CSS styles. Still, the whole cascading nature of CSS can easily destroy all encapsulation that we try to bring in place using CSS-structuring patterns.

Time for new standards

Web standards have been evolving immensely in the last couple of years. There are so many new standards, and the browser became such a big multimedia framework, that it's hard for other platforms to compete with this.

I'd even go as far as to say that web technology will actually replace other frameworks in the future, and it probably will be renamed to multimedia technology or something similar. There's no reason why we need to use different native frameworks to create user interfaces and presentations. Web technologies embed so many features that it's hard to find a reason not to use them for any kind of application. Just look at the Firefox OS or the Chrome OS, which are designed to run with web technologies. I think it's only a matter of time until more operating systems and embedded devices make use of web technologies to implement their software. This is why I believe that at some point it will be questionable whether the term *web technologies* is still appropriate or whether we should replace it with a more general term.

Although we usually just see new features appear in browsers, there is a very open and long-winded standardization process behind them. It's very important to standardize features, but this takes a lot of time, especially when people disagree about different approaches to solving problems.

Coming back to the concept of components, this is something where we really need support from web standards to break the current limitations. Fortunately, the W3C thought the same, and a group of developers started to work on specifications under the hood of an umbrella specification called *web components*.

The following topics will give you a brief overview over two specifications that also play a role in Angular 2 components. One of Angular 2's core strengths is that it acts more like a superset of web standards rather than being a complete isolated framework.

Template elements

Template elements allow you to define regions within your HTML, which will not be rendered by the browser. You can then instantiate these document fragments with JavaScript and then place the resulting DOM within your document.

While the browser is actually parsing the template content, it only does so in order to validate the HTML. Any immediate actions that the parser would usually execute will not be taken. Within the content of template elements, images will not be loaded and scripts won't be executed. Only after a template is instantiated, the parser will take the necessary actions, as follows:

```
<body>
<template id="template">
  <h1>This is a template!</h1>
</template>
```

```
</body>
```

This simple HTML example of a template element won't display the heading on your page. As the heading is inside a template element, we first need to instantiate the template and add the resulting DOM into our document:

```
var template = document.querySelector('#template');
var instance = document.importNode(template.content, true);
document.body.appendChild(instance);
```

Using these three lines of JavaScript, we can instantiate the template and append it into our document.

Shadow DOM

This part of the web components specification was the missing piece to create proper DOM encapsulation and composition. With shadow DOM, we can create isolated parts of the DOM that are protected against regular DOM operations from the outside. Also, CSS will not reach into shadow DOM automatically, and we can create local CSS within our component.

Tip

If you add a `style` tag inside shadow DOM, the styles are scoped to the root within the shadow DOM, and they will not leak outside. This enables a very strong encapsulation for CSS.

Content insertion points make it easy to control content from the outside of a shadow DOM component, and it provides some kind of an interface to pass in content.

At the time of writing this book, shadow DOM is supported by most browsers although it still needs to be enabled in Firefox.

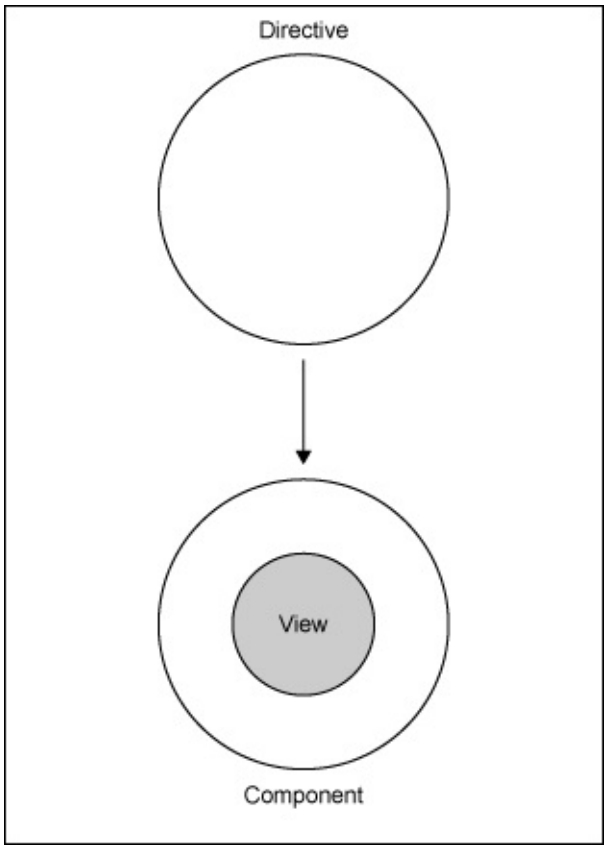
Angular's component architecture

For me, the concept of directives from the first version of Angular changed the game in frontend UI frameworks. This was the first time that I felt that there was a simple yet powerful concept that allowed the creation of reusable UI components. Directives could communicate with DOM events or messaging services. They allowed you to follow the principle of composition, and you could nest directives and create larger directives that solely consisted of smaller directives arranged together. Actually, directives were a very nice implementation of components for the browser.

In this section, we'll look into the component-based architecture of Angular 2 and how the things we've learned about components will fit into Angular.

Everything is a component

As an early adopter of Angular 2 and while talking to other people about it, I got frequently asked what the biggest difference is to the first version. My answer to this question was always the same. Everything is a component.



For me, this paradigm shift was the most relevant change that both simplified and enriched the framework. Of course, there are a lot of other changes with Angular 2. However, as an advocate of component-based user interfaces, I've found that this change is the most interesting one. Of course, this change also came with a lot of architectural changes.

Angular 2 supports the idea of looking at the user interface holistically and supporting composition with components. However, the biggest difference to its first version is that now your pages are no longer global views, but they are simply components that are assembled from other components. If you've been following this chapter, you'll notice that this is exactly what a holistic approach to user interfaces demands. No more pages but systems of components.

Tip

Angular 2 still uses the concept of directives, although directives are now really what the name suggests. They are orders for the browser to attach a given behavior to an element.

Components are a special kind of directives that come with a view.

Your first component

Keeping up the tradition, before we start building a real application together, we should write our first `hello world` component with Angular:

```
// Decorators allow us to separate declarative logic from our
// component implementation logic.
@Component({
  selector: 'hello-world',
  template: '<div>Hello {{name}}</div>'
})
class HelloWorld {
  constructor() {
    this.name = 'World';
  }
}
```

This is already a fully-working Angular 2 application. We used ECMAScript 6 classes to create the necessary encapsulation required for a component. You can also see a meta-annotation that is used to declaratively configure our component. This statement, which looks like a function call that is prefixed with an *at* symbol actually comes from the ECMAScript 7 decorator proposal.

Note

ECMAScript 7 decorators are still very experimental at the time of writing this book. For the code in this book, we actually use the TypeScript transpiler version 1.5 that already implements decorators with a slight twist to the original specification. TypeScript 1.5 is also used by the Angular 2 team to develop the core of Angular.

It's important to understand that an element can only be bound to one single component. As a component always comes with a view, there is no way that we can bind more than one component to an element. On the other hand, an element can be bound to many directives, as directives don't come with a view but they only attach behavior.

In the `Component` decorator, we need to configure everything that is relevant to describe our component for Angular. This, of course, also includes our template for the view. In the preceding example, we are specifying our template directly within JavaScript as a string. We can also use the `templateUrl` property to specify a URL where the template should be loaded from.

Now, let's enhance our example a little bit so that we can see how we can compose our application from smaller components:

```
// Using decorators we can declaratively define our component used
// to write bold text
@Component({
  selector: 'shout-out',
```

```
  template: '<strong>{{words}}</strong>'
})
class ShoutOut {
  @Input() words;
}

// This component will be our main application component that
// makes use of the above shout-out component (composition)
@Component({
  selector: 'hello-world'
  template: '<shout-out words="Hello, {{name}}!"></shout-out>',
  directives: [ShoutOut]
})
class HelloWorld {
  constructor() {
    this.name = 'World';
  }
}
```

You can see that we have now created a small component that allows us to shout out words as we like. In our *Hello World* application, we make use of this component to shout out **Hello, World!**

Tip

Every directive or component that is used inside a components view template needs to be explicitly declared in the directives property of the view annotation. Otherwise, the compiler will not recognize the directive when it encounters the element in the template.

Over the course of this book and while writing our task management application, we will learn a lot more about the configuration and implementation of components. However, before we start with this in the second chapter, we should take a look at some tools and language features that we'll use during this book.

JavaScript of the future

It was not so long ago that somebody asked me whether we should really use the bind function of ECMAScript 5.1, as then we'd probably run into browser compatibility issues. The web moves very fast, and we need to keep up the pace. We can't write code that does not use the latest features even if this would cause issues in old browsers.

The fantastic people from TC39, the technical committee that is responsible for writing the ECMAScript specification, have done a great job progressively enhancing the JavaScript language. This, and the fact that JavaScript is so flexible, allows us to use so-called polyfills and shims to make our code run in older browsers.

ECMAScript 6 (also referred to as ECMAScript 2015) was published in June 2015, exactly four years after its predecessor. There is a massive amount of new API additions as well as a whole bunch of new language features. The language features are syntactic sugar, and ECMAScript 6 can be transpiled to its previous version where it runs perfectly in older browsers. At the time of writing this book, none of the current browser versions have fully implemented ECMAScript 6, but there's absolutely no reason not to use it for production applications.

Tip

Syntactic sugar is a design approach where we evolve a programming language while not breaking backwards compatibility. This allows language designers to come up with new syntax, which enriches developer experience but does not break the web. Every new feature needs to be translatable to the old syntax. This way, so-called transpilers can be used to convert code to older versions.

I speak JavaScript, translate, please!

While compilers compile from a higher-level language to a lower-level language, a transpiler or transcompiler acts more like a converter. It is a source-to-source compiler that translates code to run in a different interpreter.

Recently, there's a real battle among new languages that are transpiled to JavaScript and can run in the browser. I used Google Dart for quite some time, and I must admit, I really loved the language features. The problem with nonstandardized languages is that they depend heavily on community adoption and the hype. Also, it's almost certain that they will never run natively within the browser. This is also the reason why I prefer standard JavaScript, and the JavaScript of the future uses transpilers that allow me to do this.

Some people argue that transpilers introduce code that does not run very performant and, therefore, recommend that you do not use ECMAScript 6 and transpilers at all. I don't agree with this because of many reasons. Usually, this is about performance in micro or even nanosecond areas where this often really does not matter for most applications.

I don't say performance doesn't matter, but performance needs to always be discussed within a context. If you're trying to optimize a loop within your application by reducing processing time from 10 microseconds to five microseconds where you'd never iterate over more than 100 items, then you're probably spending your time on the wrong things.

Also, a very important fact is that transpiled code is designed by people who understand micro performance optimization much better than I do, and I'm sure their code runs faster than mine. On top of this, a transpiler is probably also the right place where you'd want to do performance optimization because this code is automatically generated and you don't lose maintainability of your code through performance quirks.

I'd like to quote Donald Knuth here and say that premature optimization is the root of all evil. I really recommend that you read his paper on this topic (Donald Knuth, December 1974, *Structured Programming with go to Statements*). Just because the `goto` statements got banished from all modern programming languages, it doesn't mean this is less of a good read.

Later on in this chapter, you'll learn about tools that help you use transpilers easily within your project, and we'll take a look at the decisions and directions Angular went with their source code.

Let's look at a few language features that come with ECMAScript 6 and make our life much easier.

Classes

Classes were among one the most requested features in JavaScript, and I was one of the people voting for it. Well, coming from an OOP background and being used to organizing everything within classes, it was hard for me to let go. Although, after working with modern JavaScript for some time, you'll reduce their use to the bare minimum and to exactly what they are made for—inheritance.

Classes in ECMAScript 6 provide you with syntactic sugar to deal with prototypes, constructor functions, super calls, and object property definitions in a way that you have the illusion that JavaScript could be a class-based OOP language:

```
class Fruit {
  constructor(name) { this.name = name; }
}
const apple = new Fruit('Apple');
```

As we learned in the previous topic about transpilers, ECMAScript 6 can be de-sugared to ECMAScript 5. Let's take a look at what a transpiler would produce from this simple example:

```
function Fruit(name) { this.name = name; }
var apple = new Fruit('Apple');
```

This simple example can easily be built using ECMAScript 5. However, once we use the more

complex features of class-based object-oriented languages, the de-sugaring gets quite complicated.

ECMAScript 6 classes introduce simplified syntax to write class member functions (static functions), the use of the super keyword, and inheritance using the `extends` keyword.

If you would like to read more about the features in classes and ECMAScript 6, I highly recommend that you read the articles of Dr. Axel Rauschmayer (http://www.2ality.com/).

Modules

Modules provide a way to encapsulate your code and create privacy. In object-oriented languages, we usually use classes for this. However, I actually believe this is an antipattern rather than a good practice. Classes should be used where inheritance is desired and not just to structure your code.

I'm sure that you've encountered a lot of different module patterns in JavaScript already. One of the most popular ones that creates privacy using a function closure of an **immediately invoked function expression** (**IIFE**) is probably the revealing module pattern. If you'd like to read more about this and may be other great patterns, I recommend the book, *Learning JavaScript Design Patterns*, by Addy Osmani.

Within ECMAScript 6, we can now use modules to serve this purpose. We simply create one file per module, and then we use the import and export keywords to connect our modules together.

Within the ECMAScript 6 module specification, we can actually export as many things as we like from each module. We can then import these named exports from any other module. We can have one default export per module, which is especially easy to import. Default exports don't need to be named, and we don't need to know their name when importing them:

```
import SomeModule from './some-module.js';
var something = SomeModule.doSomething();
export default something;
```

There are many combinations on how to use modules. We will discover some of these together while working on our task management application during the upcoming chapters. If you'd like to see more examples on how to use modules, I can recommend the Mozilla Developer Network documentation (https://developer.mozilla.org) on the `import` and `export` keywords.

Template strings

Template strings are a very simple, but they are an extremely useful addition to the JavaScript syntax. They serve three main purposes:

- Writing multiline strings

- String interpolation
- Tagged template strings

Before template strings, it was quite verbose to write multiline strings. You needed to concatenate pieces of strings and append a new-line character yourself to the line endings:

```
const header = '<header>\n' +
  '  <h1>' + title + '</h1>\n' +
  '</header>';
```

Using template strings, we can simplify this example a lot. We can write multiline strings, and we can also use the string interpolation functionality for our title variable that we used to concatenate earlier:

```
const header = '
  <header>
    <h1>${title}</h1>
  </header>
`;
```

Note the back ticks instead of the previous single quotes. Template strings are always written between back ticks, and the parser will interpret all characters in between them as part of the resulting string. This way, the new-line characters present in our source file will also be part of the string automatically.

You can also see that we have used the dollar sign, followed by curly brackets to interpolate our strings. This allows us to write arbitrary JavaScript within strings and helps a lot while constructing HTML template strings.

You can read more about template strings on the Mozilla Developer Network.

ECMAScript or TypeScript?

TypeScript was created in 2012 by Anders Hejlsberg with the intention to implement the future standard of ECMAScript 6 but also to provide a superset of syntax and features that was not part of the specification.

There are many features in TypeScript as a superset to the ECMAScript 6 standard, including, but not limited to the following:

- Optional static typing with type annotations
- Interfaces
- Enum types
- Generics

It's important to understand that all of the features that TypeScript provides as a superset are optional. You can write pure ECMAScript 6 and not take advantage of the additional features that TypeScript provides. The TypeScript compiler will still transcompile pure ECMAScript 6

code to ECMAScript 5 without any errors.

Note

Most of the features that are seen in TypeScript are actually already present in other languages, such as Java and C#. One goal of TypeScript was to provide language features that support workflows and better maintainability for large-scale applications.

The problem with any nonstandard language is that nobody can tell how long the language will be maintained and how fast the momentum of the language will be in the future. In terms of support, the chances are high that TypeScript, with its sponsor Microsoft, will actually have a long life. However, there's still no guarantee that the momentum and trend of the language will keep moving at a reasonable pace. This problem does obviously not exist for standard ECMAScript 6 because it's what the web of the future is made of and what browsers will speak natively.

Still, there are valid reasons to use the extended features of TypeScript if you'd want to address the following concerns that clearly outweigh the negative implications of an uncertain future in your project:

- Large applications that undergo a huge amount of changes and refactoring
- Large teams that require a strict governance while working on code

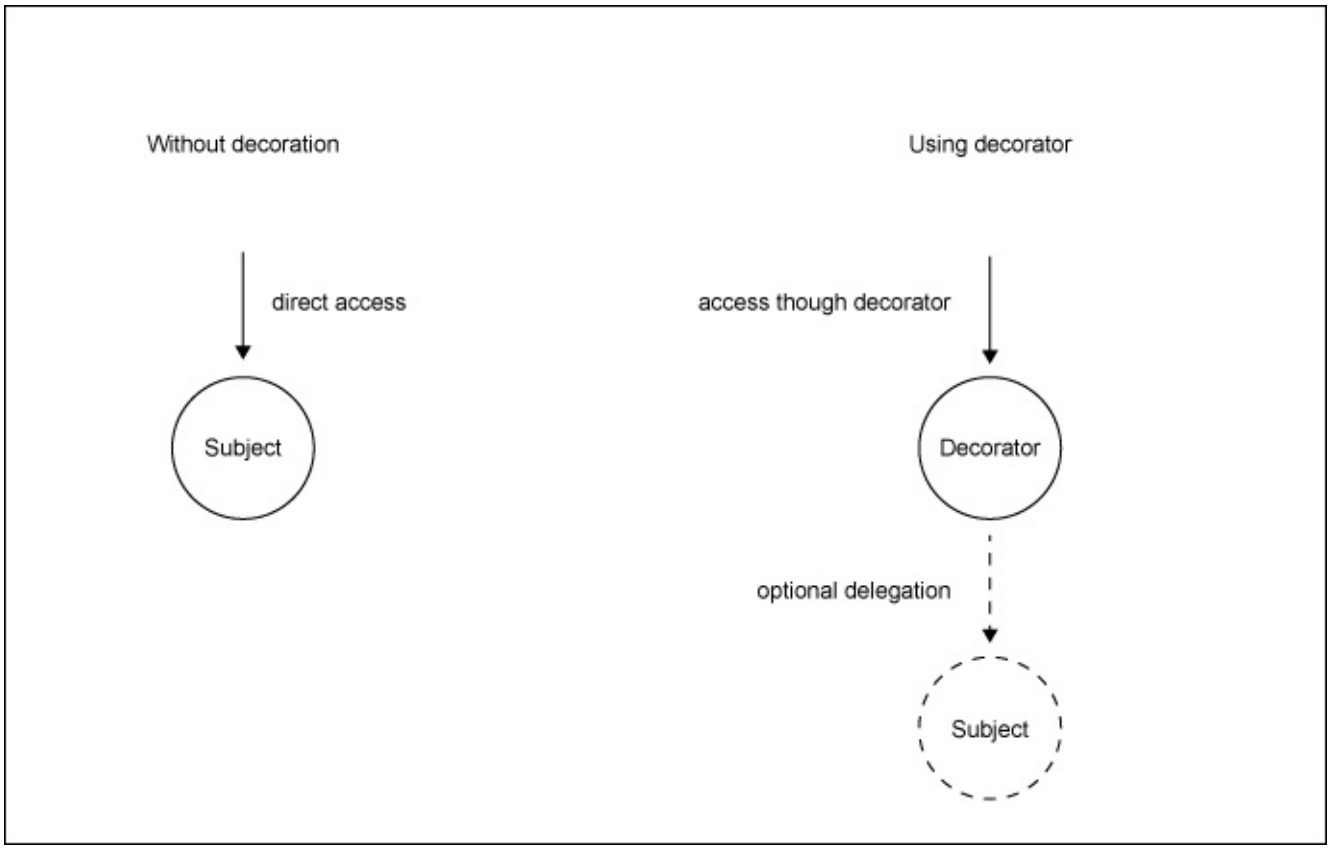
In this book, we'll use a TypeScript compiler, but we will work with standard ECMAScript 6 code with one exception that is covered in the next topic about decorators.

Decorators

Decorators are not part of the ECMAScript 6 specification, but they were proposed to the ECMAScript 7 standard for 2016. They provide us with a way to decorate classes and properties during design time. This allows a developer to use meta-annotations while writing classes, and declaratively attach functionality to the class and its properties.

Decorators are named after the decorator pattern that was initially described in the book *Design Patterns: Elements of Reusable Object-Oriented Software* of Erich Gamma and his colleagues, also known as the **Gang of Four** (**GoF**).

The principle of decoration is that an existing procedure is intercepted and the decorator has the chance to either delegate, provide an alternative procedure, or do a mix from both.



Visualization of decoration in a dynamic environment with the example of a simple access procedure

Decorators in ECMAScript 7 can be used to annotate classes and class properties. Note that this also includes class methods, as class methods are also properties of the class prototype object. Decorators get defined as regular functions, and they can be attached to classes or class properties with the *at* symbol. Our decorator function will then be called with contextual information about the location of inclusion every time that the decorator is placed.

Let's take a look at a simple example that illustrates the use of a decorator:

```
function logAccess(obj, prop, descriptor) {
  const delegate = descriptor.value;
  descriptor.value = function() {
    console.log(`${prop} was called!`);
    return delegate.apply(this, arguments);
  };
}

class MoneySafe {
  @logAccess
  openSafe() {
    this.open = true;
  }
}
```

```
const safe = new MoneySafe();
safe.openSafe(); // openSafe was called!
```

We have created a `logAccess` decorator that will log all function calls that are tagged with the decorator. If we look at the `MoneySafe` class, you can see that we have decorated the `openSafe` method with our `logAccess` decorator.

The `logAccess` decorator function will be executed for each annotated property within our code. This enables us to intercept the property definition of the given property. Let's take a look at the signature of our decorator function. Decorator functions that are placed on class properties will be called with the target object of the property definition as a first parameter. The second parameter is the actual property name that is defined, followed by the last parameter, which is the descriptor object that is supposed to be applied to the object.

The decorator gives us the opportunity to intercept the property definition. In our case, we use this ability to exchange the descriptor value (which is the annotated function) with a proxy function that will log the function call before calling the origin function (delegation). For simplification purposes, we've implemented a very simple yet incomplete function proxy. For real-world scenarios, it would be advisable to use a better proxy implementation, such as the ECMAScript 6 proxy object.

Decorators are a great feature to leverage aspect-oriented concepts and declaratively add behavior to our code at design time.

Let's look at a second example where we use an alternative way to declare and use decorators. We can treat decorators like function expressions where our decorator function is rewritten as a factory function. This form of usage is especially useful when you need to pass along configuration to the decorator, which is made available in the decorator factory function:

```
function delay(time) {
  return function(obj, prop, descriptor) {
    const delegate = descriptor.value;
    descriptor.value = function() {
      const context = this;
      const args = arguments;
      return new Promise(function(success) {
        setTimeout(function() {
          success(delegate.apply(context, arguments));
        }, time);
      });
    };
  };
}

class Doer {
  @delay(1000)
  doItLater() {
    console.log('I did it!');
  }
```

```
}

const doer = new Doer();
doer.doItLater(); // I did it! (after 1 second)
```

We have now learned how ECMAScript 7 decorators can help you to write declarative code that has an aspect-oriented twist to it. This simplifies development a lot because we can now think of behavior that we add to our classes during design time when we actually think about the class as a whole and write the initial stub of the class.

Decorators in TypeScript are slightly different than the decorators from ECMAScript 7. They are not limited to classes and class properties, but they can also be placed on parameters within the class methods. This allows you to annotate function parameters, which can be useful in some cases:

```
class TypeScriptClass {
  constructor(@ParameterDecorator() param) {}
}
```

Angular uses this feature to simplify dependency injection on class constructors. As all directive, component, and service classes get instantiated from Angular dependency injection and not by us directly, these annotations help Angular find the correct dependencies. For this use-case, function parameter decorators actually make a lot of sense.

Note

Currently, there are still issues with the implementation of decorators on class method parameters, which is also why ECMAScript 7 does not support it. As this feature is crucial to build an Angular 2 application, we'll use the TypeScript compiler to transpile the code of our application. This is the only TypeScript-specific feature that we'll use in this book.

Tools

In order to make use of all these future technologies, we need some tools to support us. We were already talking about ECMAScript 6 and decorators, where we actually prefer TypeScript decorators, as they support the function parameter decorators that are used by Angular 2. Although the ECMAScript 6 syntax supports modules, we still need some sort of a module loader that will actually load the required modules in the browser or help us generate an executable bundle.

Node.js and NPM

Node.js is JavaScript on steroids. Initially, a fork of the V8 JavaScript engine from the Google Chrome browser, Node.js was extended with more functionality, specifically to make JavaScript useful on the server-side. File handling, streams, system APIs, and a huge ecosystem of user-generated packages are just some of the facts that make this technology an outstanding partner for your web development.

The node package manager, NPM, is a door to over 200,000 packages and libraries that help you build your own application or library. The Node.js philosophy is very similar to the UNIX philosophy, where packages should stay small and sharp, but they should use composition to achieve greater goals.

To build our application, we will rely on Node.js as the host for the tools that we're going to use. We should, therefore, make sure that we install Node.js on our machine so that we are prepared for the next chapter, where we start to craft our task management application.

Note

You can get Node.js from their website at https://nodejs.org, and it should be a breeze to install this on any kind of operating system by following the instructions on their website.

Once you've installed Node.js, we can perform a simple test to check whether everything is up and running. Open a terminal console and execute the following command:

```
node -e "console.log('Hello World');"
```

SystemJS and JSPM

There are many module formats and module loaders out there, but there's one that rules them all in my opinion. SystemJS is built on top of an ES6 module loader polyfill and, therefore, moves very close to an upcoming standard. I strongly believe in standardization and, therefore, prefer SystemJS over other module loaders, such as RequireJS, Browserify, or webpack. We should stop using libraries where possible and rely on polyfills that make our browser capable of running the code of the future.

SystemJS is a universal module loader that is capable of loading many different module formats, such as AMD, CommonJS, and ECMAScript 6, and it also supports a very flexible shiming mechanism to modularize global JavaScript.

SystemJS also supports the most popular transpilers, including ECMAScript 6 and TypeScript. This means that you can actually load ECMAScript 6 code directly in your browser, where it's transpiled by SystemJS in runtime. This is great during development, especially because you're allowed to load modules from any location, including remote HTTP locations, such as GitHub or the NPM repository.

JSPM

The JavaScript package manager is not just another package manager for JavaScript. This is basically a mediator and manager for SystemJS that helps you look up packages from package repositories, such as Bower or NPM, and it creates the necessary configuration for SystemJS. JSPM is written in Node.js and does not come with its own remote package repository. As SystemJS needs URLs and module mappings to know where to load modules from, JSPM is your tool to create this necessary configuration and simplify package installation.

Getting started with JSPM

Let's create a simple application together using JSPM. First of all, we need to install two global modules with NPM. Besides JSPM, we'll also install a tool called live-server, which will help us during development by providing an HTTP server that serves static files. It also has file change detection built-in, and it will reload your browser automatically once a file change has been detected. This provides a very short feedback loop and makes development a very fast process:

1. Run the following command on your command line:

   ```
   npm install -g jspm live-server
   ```

   ### Tip

   Note that on UNIX-like systems, such as Linux or Mac OS X, it's sometimes required to run NPM as a super user. It's also recommended that you use the **Node Version Manager** (**NVM**) to get around those issues (https://github.com/creationix/nvm).

2. After installing JSPM and the `live-server` package, we can go ahead and create our first application using JSPM.
3. Create a new directory for the application, and open a terminal console within this directory.
4. You can now execute the following command on your terminal console to install JSPM locally and initialize a new JSPM project:

```
npm install jspm --save-dev
jspm init
```

5. JSPM will start a wizard that guides you though the initialization steps. You can answer all questions with the default answer (just hit *Enter*) except for the question about which transpiler you'd like to use that you should answer with TypeScript.
6. After JSPM installs all the necessary packages, we can go ahead and create our `index.html` file. Navigate to your project folder and create a new file, `index.html`, in your favorite editor:

```
<!doctype html>
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
  System.import('main.js');
</script>
```

7. This very minimalistic HTML is already the foundation for our JSPM Hello World application. After including the `SystemJS` library and the `config.js` file that was generated by JSPM, we only need to bootstrap our application by telling `SystemJS` which file to import.
8. Before we create our main application file, we will quickly install jQuery as a package, just to demonstrate how easily third-party libraries can be installed and used with `SystemJS` and JSPM:

```
jspm install jquery
```

9. After installing jQuery, we can go ahead and create our `main.js` file inside of the application folder:

```
import $ from 'jquery';

class HelloWorld {
  constructor() {
    $(document.body).append('<h1>Hello World!</h1>');
  }
}
const helloWorld = new HelloWorld();
```

10. In order to run this example in the browser, we can now start our live server with the following command executed inside of the application folder:

```
live-server
```

After following the preceding steps, you should have a working example with ECMAScript 6 and SystemJS using the TypeScript transpiler. Using LiveReload, your browser should automatically open and display our Hello World application. You can also try now to modify the code a bit and change the sentence that is written to the DOM. You'll notice that once you save your changes, the browser will immediately reload the page.

Summary

In this chapter, we looked at a component-based approach to structure user interfaces, and we talked about the necessary aspects of its background to understand why we are moving in this direction with the web standard and frameworks, such as Angular. We also ensured that we are prepared with all the technology that we will use in the upcoming chapters in this book. You created your first simple example using JSPM, SystemJS, ECMAScript 6, and the TypeScript transpiler. Now, we are ready to start building our task-management system using a component-based architecture to its full potential.

In the next chapter, we're going to start building our task management application using Angular 2 components. We'll look at the initial steps that are required to create an Angular 2 application from scratch and flesh out the first few components in order to build a task list.

Chapter 2. Ready, Set, Go!

In this chapter, we will start building our task management application. We'll jump right into the core of the application and create the initial components required to manage a simple task list. In the process of going through this chapter, you'll learn about the following topics:

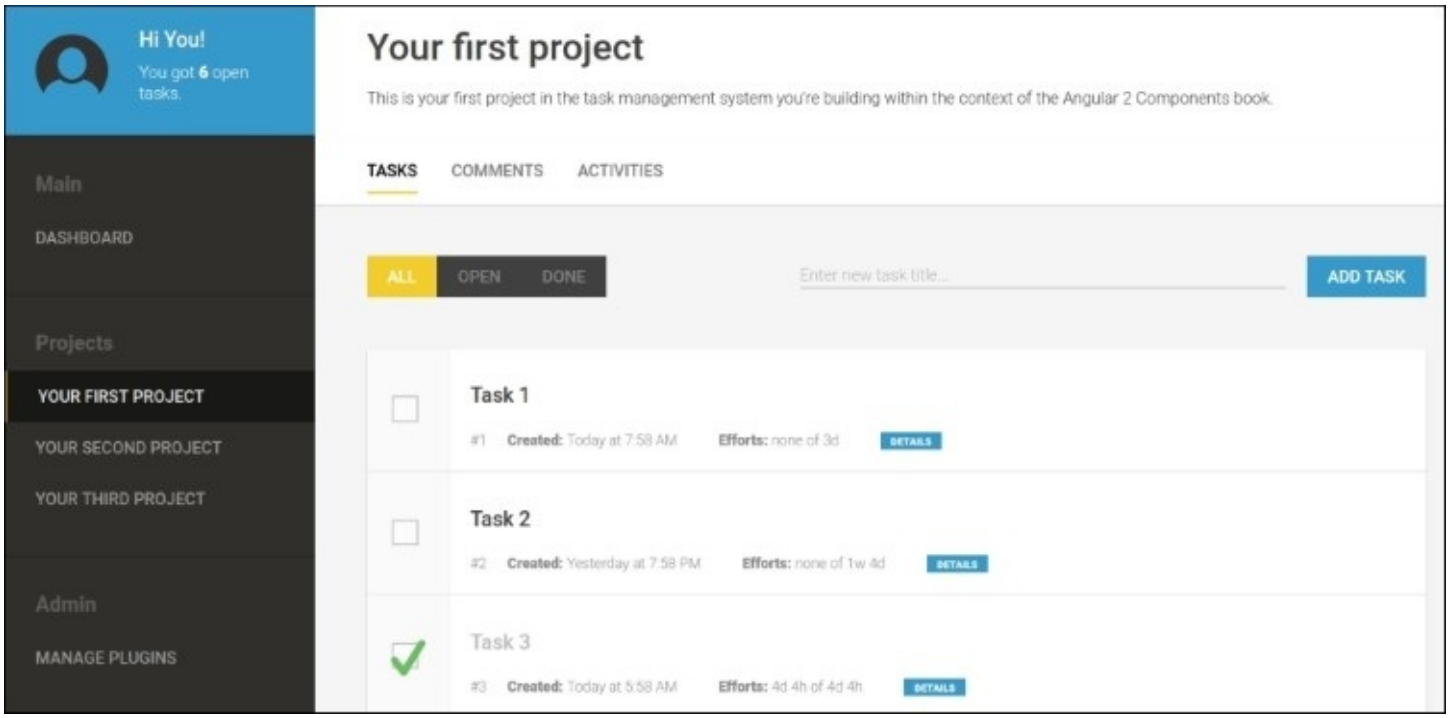
- Bootstrapping an Angular application using a main component
- Component input and output
- Host property binding
- Styling and view encapsulation
- Importing HTML templates using the SystemJS text loader
- Using `EventEmitter` to emit custom events
- Two-way data binding
- Component life cycle

Managing tasks

After picking up the basics from the previous chapter, we will now go on and create a task management application together in the upcoming chapters. You'll learn about some concepts during the chapters and then use them with practical examples. You'll also learn how to structure an application using components. This begins with the folder structure and ends with setting up the interaction between components.

Vision

The task management application should enable users to manage tasks easily and help them organize small projects. Usability is the central aspect of any application; therefore, you'll need to design a modern and flexible user interface that will support the user.



A preview of the task management application we are going to build

Our task management application will consist of components that will allow us to design a platform providing a great user experience for the purpose of managing tasks. Let's define the core features of our application:

- Managing tasks within multiple projects and providing a project overview
- Simple scheduling as well as a time-and-effort-tracking mechanism
- Overviewing the dashboard using graphical charts
- Tracking activities and providing a visual audit log
- A simple commenting system that would work across different components

The task management application is the main example in this book. Therefore, the building blocks within the book should only contain the code that is relevant to the theme of the book. Of course, other than components, an application needs other functionalities, such as visual design, data, session management, and other important parts, to work. While the required code for each chapter can be downloaded online, we'll only discuss the code relevant to the topics learned within the book.

Starting from scratch

Let's start out by creating a new folder called `angular-2-components` in order to create our application:

1. Open a console window inside our newly created folder and run the following command to initialize a new Node.js project:

   ```
   npm init
   ```

2. Finish the initialization wizard by confirming all the steps with the *Enter* key (default settings).
3. Since we're using JSPM to manage our dependencies, we need to install it as a project Node.js package:

   ```
   npm install jspm --save-dev
   ```

4. Let's also initialize a new JSPM project within our project folder. Be sure to use the default settings (just hit the *Enter* key) for all settings, except for the step where you are asked which transpiler you'd like to use. Enter TypeScript at this stage:

   ```
   jspm init
   ```

5. We'll now use JSPM to install the relevant Angular 2 packages into our project as dependencies. We'll also install a SystemJS loader plugin to load text files as modules. We'll provide some details around this later on:

   ```
   jspm install npm:@angular/core npm:@angular/common npm:@angular/compiler
   npm:@angular/platform-browser-dynamic npm:rxjs text
   ```

Let's examine what we've been creating so far by using the NPM and JSPM command-line tools.

The `package.json` file is our Node.js configuration file that we're using as the base to work with JSPM (the package manager) and SystemJS (the module loader with transpiler). If you check out the `package.json` file, you will see an additional section for JSPM dependencies:

```
  "jspm": {
    "dependencies": {
      "@angular/common": "npm:@angular/common@2.0.0-rc.1",
      "@angular/compiler": "npm:@angular/compiler@2.0.0-rc.1",
      "@angular/core": "npm:@angular/core@2.0.0-rc.1",
      "@angular/platform-browser-dynamic": "npm:@angular/platform-browser-
dynamic@2.0.0-rc.1",
      "text": "github:SystemJS/plugin-text@0.0.7"
    },
    "devDependencies": {
      "typescript": "npm:typescript@1.8.10",
    }
  }
```

Let's take a quick look at the dependencies we have installed using JSPM and their purpose:

Package	Description
`@angular/core`	This is the core package of Angular 2, hosted on NPM. If you remember from Chapter 1, *Component-Based User Interfaces*, JSPM is only a broker, and it delegates to other package repositories. The core package contains all Angular-core modules, such as the `@Component` decorator, change detection, dependency injection, and more.
`@angular/common`	The Angular `common` package provides us with base directives, such as `NgIf` and `NgFor`. It also contains all the base pipes and the directives that are used to control forms.
`@angular/compiler`	The compiler package contains all the artifacts required to compile view templates. Angular not only provides the ability to precompile templates to gain faster booting time, but it also uses the compiler at runtime to convert text templates into compiled templates. This package is required if we're compiling templates at runtime.
`@angular/platform-browser-dynamic`	This package includes the bootstrapping functionality that will help us start our application. The bootstrap initiated by the platform-browser-dynamic package is dynamic in the sense of compiling templates at runtime.
`typescript`	This development dependency is the TypeScript transpiler for SystemJS. It transpiles our ECMAScript 6 and TypeScript code to ECMAScript 5, from where it can run in the browser.
`text`	This SystemJS loader supports the loading of text files in the form of JavaScript strings. This is especially useful if you like to load HTML templates and avoid asynchronous requests.

Our main entry point for displaying our application within the browser is our index site. The `index.html` file completes the following five actions:

- Loading ECMAScript 6 polyfill es6-shim from a CDN. This script is required to make sure the browser understands the latest ECMAScript 6 APIs.
- Loading the Angular 2 polyfills required by the framework. This includes various patches for the browser that are required to run an Angular 2 application. It's important

to load these polyfills before we load any other code within our application.
- Loading SystemJS and the SystemJS `config.js` file that contains the mapping information generated by JSPM.
- Using the `System.import` function to load and execute the main entry point, which is our `boostrap.js` file.

Let's create a new `index.html` file within the root folder of our project:

```
<!doctype html>
<html>
<head lang="en">
  <title>Angular 2 Components</title>
</head>
<body>

<script src="https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.35.0/es6-
shim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/2.0.0-
beta.15/angular2-polyfills.js"></script>
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script>
  System.import('lib/bootstrap.js');
</script>
</body>
</html>
```

Let's move on to our application component. You can think of it as the outermost component of your application. It's the main component in that it represents your whole application. Every application needs one and just one main component. This is where your component tree has its roots.

We'll name our main component `App` because it represents our whole application. Let's go ahead and create the component within a new `lib` folder in our project folder. Create a file, `app.js`, with the following content:

```
// We need the Component annotation as well as the
// ViewEncapsulation enumeration
import {Component, ViewEncapsulation} from '@angular/core';

// Using the text loader we can import our template
import template from './app.html!text';

// This creates our main application component
@Component({
  // Tells Angular to look for an element <ngc-app> to create this
  // component
  selector: 'ngc-app',
  // Let's use the imported HTML template string
  template,
  // Tell Angular to ignore view encapsulation
  encapsulation: ViewEncapsulation.None
})
```

```
export class App {}
```

There's nothing different here from what we already know about structuring a component, something that we learned in the previous chapter. However, there are two main differences here compared to how we created the components before. If you look at how we configured the `template` property, you could tell that we didn't write the HTML template directly within the JavaScript file inside the ECMAScript 6 template strings. Instead, we're going to load the template into a JavaScript string using the text loader plugin in SystemJS. We can just load any text file from the file system by appending `!text` to our regular ECMAScript 6 imports:

```
import template from './app.html!text';
```

This will load the file, `app.html`, from the current directory and make a default export with its content as a string.

The second difference is that we're using `ViewEncapsulation` to specify how Angular should handle view encapsulation. Angular has three ways, to handle view encapsulation, which provides different levels of granularity and has their own pros and cons. They are as follows:

Encapsulation type	**Description**
`ViewEncapsulation.Emulated`	If a component is set to emulated view encapsulation, it will emulate style encapsulation by attaching the generated attributes to the component element and modifying CSS selectors to include these attribute selectors. This will enable certain forms of encapsulation, although the outer styles can still leak into the component if there are other global styles. This view encapsulation mode is the default mode, if not specified otherwise.
`ViewEncapsulation.Native`	Native view encapsulation is supposed to be the ultimate goal of the view encapsulation concept within Angular. It makes use of Shadow DOM, as described in the previous chapter, to create an isolated DOM for the whole component. This mode depends on the browser to support Shadow DOM natively, and therefore, can't always be used. It's also important to note that global styles will no longer be respected and local styles need to be placed within the component in inline style tags (or use the `styles` property on the component annotation).
	This mode tells Angular not to provide any template or style encapsulation. Within our application, we mainly rely on

`ViewEncapsulation.None`	styles coming from a global CSS; therefore, we use this mode for most of the components. Neither Shadow DOM, nor attributes will be used to create style encapsulation; we can simply use the classes specified within our global CSS file.

As this component is now relying on a template to be loaded from the file system, we need to create the `app.html` file in the `lib` folder with some initial content:

```
<div>Hello World!</div>
```

For the time being, that's everything we put in our template. Our directory should look similar to this:

```
angular-2-components
├── node_modules/
├── jspm_packages/
├── config.js
├── index.html
├── lib
│   ├── app.html
│   └── app.js
└── package.json
```

Now that we have created our main application component, we can add the component's host element to our `index.html` file:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <title>Angular 2 Components</title>
</head>
<body>
<ngc-app></ngc-app>
...
```

Bootstrapping

The `index.html` file will load the `bootstrap.js` module using SystemJS in an inline `script` tag. It's a best practice to have a main entry point for your scripts when working with SystemJS. Our `bootstrap.js` file is responsible for loading all the necessary JavaScript dependencies for our application as well as bootstrapping the Angular framework.

We can go ahead and bootstrap our Angular application by providing our main application component, `App`. We need to import the `bootstrap` function from the `angular2` module. We can then import our `App` component and call the `bootstrap` function, passing it as parameter:

```
// Import Angular bootstrap function
import {bootstrap} from '@angular/platform-browser-dynamic';
// Import our main app component
import {App} from './app';
// We are bootstrapping Angular using our main application
// component
bootstrap(App);
```

Running the application

The code we've produced so far should now be in a state where we can run it. Before we run our code using the live-server module, let's ensure we have all the files ready. At this stage, our directory should look something like this:

```
angular-2-components
├── jspm_packages/
├── node_modules/
├── config.js
├── index.html
├── lib
│   ├── app.html
│   ├── app.js
│   └── bootstrap.js
└── package.json
```

Now let's start live server to start a server and a browser with live reload. For this, we need to simply execute the following command on the command line within our project folder:

```
live-server
```

If everything goes well, you will have an open web browser that shows **Hello World!**.

Recap

Let's recap what we have done so far:

1. We initialized a new project using NPM and JSPM and installed the Angular dependencies using JSPM.
2. We created our main application component in `app.js`.
3. We also created a `bootstrap.js` script to include the Angular framework boot of our application.
4. We added our component to the `index.html` file by including an element that matches our component selector property.
5. Finally, we used live server to start a basic web server and launch a web browser.

Creating a task list

Now that we have our main application component set up, we can go on and flesh out our task application. The second component that we're going to create will be responsible for listing tasks. Following the concept of composition, we'll create a `task-list` component as a subcomponent of our main application component.

Let's create a folder within the `lib` folder called `task-list` and a new JavaScript file called `task-list.js`, where we will write our component code:

```
import {Component, ViewEncapsulation} from '@angular/core';
import template from './task-list.html!text';

@Component({
  selector: 'ngc-task-list',
  // The host property allows us to set some properties on the
  // HTML element where our component is initialized
  host: {
    class: 'task-list'
  },
  template,
  encapsulation: ViewEncapsulation.None
})
export class TaskList {
  constructor() {
    this.tasks = [
      {title: 'Task 1', done: false},
      {title: 'Task 2', done: true}
    ];
  }
}
```

We've created a very simple `task-list` component that has a list of tasks stored internally. This component will be attached to HTML elements that match the CSS element selector `ngc-task-list`.

Now let's create a view for this component to display the tasks. As you can see from the import within the component JavaScript file, we are looking for a file called `task-list.html`:

```
<div *ngFor="let task of tasks" class="task">
  <input type="checkbox" [checked]="task.done">
  <div class="task__title">{{task.title}}</div>
</div>
```

We use the `NgFor` directive to repeat the `<div>` element with the class task for as many tasks as we have in the task list of our component. The `NgFor` directive in Angular will create a template element from its underlying content and instantiate as many elements from the template as the expression evaluates to. We currently have two tasks in our `task-list` component, so this will create two instances of our template.

Your folder structure inside the `lib` folder should now look similar to this:

```
angular-2-components
└── lib
    ├── app.html
    ├── app.js
    ├── bootstrap.js
    └── task-list
        ├── task-list.html
        └── task-list.js
```

All that's left to do in order to make our task list work is the inclusion of the `task-list` component within the main application component. We can go ahead and modify our `app.js` file and add the following line on top of it:

```
import {TaskList} from './task-list/task-list';
```

As we want to add the `task-list` component to our main application view template, we also need to make sure that Angular knows about the component when compiling the view. For this, we need to add the `directives` property to our main application component within the `app.js` file and include our imported `TaskList` component class within the list of directives:

```
...
  // Tell Angular to ignore view encapsulation
  encapsulation: ViewEncapsulation.None,
  directives: [TaskList]
})
...
```

Finally, we need to include the host element of our `task-list` component in the template of the main application, which is located within the `app.html` file:

```
<ngc-task-list></ngc-task-list>
```

These were the last changes we needed to make in order to make our `task-list` component work. To view your changes, you can start the live server by executing the `live-server` command within your `angular-2-components` directory.

Recap

Let's look at what we have done in the previous building block. We achieved a simple listing of tasks within an encapsulated component by following these steps:

1. We created the component JavaScript file that contains the logic of our component.
2. We created the component's view within a separate HTML file.
3. We included the component class within the configuration of our main application component.
4. We included the component HTML element within our main application view template.

The right level of encapsulation

Our task list is displayed correctly and the code we used to achieve this looks quite okay. However, if we want to follow a better approach for composition, we should rethink the design of our `task-list` component. If we draw a line at enlisting the task list responsibilities, we would come up with things such as listing tasks, adding new tasks to the list, and sorting and filtering the task list; however, operations are not performed on an individual task itself. Also, rendering the task itself falls outside of the responsibilities of the task list. The `task-list` component should only serve as a container for tasks.

If we look at our code again, we will see that we're violating the single responsibility principle and rendering the whole task body within our `task-list` component. Let's take a look at how we can fix this by increasing the granularity of the encapsulation.

The goal now is to do a code refactoring exercise, also known as extraction. We are pulling our task's relevant template out of the task list template and creating a new component that encapsulates the tasks.

For this, we need to create a new sub folder within the `task-list` folder called `task`. Within this folder, we will create a template file with the name `task.html`:

```
<input type="checkbox" [checked]="task.done">
<div class="task__title">{{task.title}}</div>
```

The content of our new `task.html` file is pretty much the same as what we already have within our `task-list.html` template. The only difference is that we will now refer to a new model called `task`.

Now, within the `task` folder, let's create the JavaScript file, `task.js`, which will contain the controller class of our component:

```
import {Component, Input, ViewEncapsulation} from '@angular/core';
import template from './task.html!text';

@Component({
  selector: 'ngc-task',
  host: {
    class: 'task'
  },
  template,
  encapsulation: ViewEncapsulation.None
})
export class Task {
  // Our task model can be attached on the host within the view
  @Input() task;
}
```

In the previous chapter of this book, we spoke about encapsulation and the preconditions to

establish a clean encapsulation for UI components. One of these preconditions is the possibility to design proper interfaces in and out of the component. Such input and output methods are necessary to make the component work within compositions. That's how a component will receive and publish information.

As you can see from our task component implementation, we are now building such an interface using the `@Input` annotation on a class instance field. In order to use this annotation, we will first need to import it from the angular core module.

Input properties in Angular allow us to bind the expressions in our templates to class instance fields on our components. This way, we can pass data from the outside of the component to the component inside, using the components template. This can be thought of as an example of one-way binding, from the view to the component.

If we're using property binding on a regular DOM property, Angular will create a binding of the expression directly to the element's DOM property. We're using such a type of binding to bind the task completed flag to the `checked` property of the checkbox's `input` element:

Usage	**Description**
`@Input() inputProp;`	This allows us to bind the `inputProp` attribute to the component element within the parent component. Angular assumes that the attribute of the element has the same name as that of the `input` property.
`@Input('inp') inputProp;`	You can also override the name of the attribute that should be mapped to this input. Here, the `inp` attribute of the component's HTML element is mapped to the component's input property, `inputProp`.

The last missing piece to use our newly created task component is the modification of the existing template of the task list.

We include the task component within our task list template by using an `<ngc-task>` element, as specified in the selector within our task component. Also, we create a property binding on the task element. There, we pass the `task` object from the current `NgFor` iteration to the `task` input of the `task` component. We need to replace all the existing content in the `task.html` file with the following lines of code:

```
<ngc-task *ngFor="let task of tasks"
          [task]="task"></ngc-task>
```

In order to make our `task-list` component recognize the task component element, we need to

add it to the `task-list` component's `directives` property within the `task-list.js` file:

```
...
import {Task} from './task/task';

@Component({
  ...
  directives: [Task]
})
...
```

Congratulations! You've successfully refactored your task list by extracting the task into its own component and have established a clean encapsulation. Also, we can now say that our task list is a composition of tasks.

If you think about maintainability and reusability, this was actually a very important step in the process of building our application. You should constantly look out for such encapsulation opportunities, and if you feel something could be arranged into multiple subcomponents, you should probably go for it. Of course, you can also overdo this. There's simply no golden rule to determine what granularity of encapsulation is the right one.

Tip

The right granularity of encapsulation for a component architecture always depends on the context. My personal tip here is to use known principles from OOP, such as single responsibility, to lay the groundwork for a good design of your component tree. Always make sure your components are only doing things that they are supposed to do as their names suggest. A task list has the responsibility of listing tasks and providing some filters or other controls for the list. The responsibility of operating on individual task data and rendering the necessary view clearly belongs to a task component and not the task list.

Recap

In this building block, we cleaned up our component tree and established clean encapsulation using subcomponents. Then, we set up the interfaces provided by Angular using input bindings. We performed these actions by following the ensuing steps:

1. We created a task subcomponent.
2. We used the task subcomponent with the `task-list` component.
3. We used input bindings and DOM element property bindings to establish one-way data binding in the task component.

Input generates output

Our task list looks nice already, but it would be quite useless if the user is unable to add new tasks to the list. Let's create a component for entering new tasks together. As this component belongs to the `task-list` component, we're going to create a new folder called `enter-task` within the `task-list` folder. The responsibilities of this component will be to handle all the UI logic necessary for entering a new task.

Using the same naming convention as with the rest of our components, let's create a file called `enter-task.html` to store the template of our component:

```
<input type="text" class="enter-task__title-input"
       placeholder="Enter new task title..."
       #titleInput>
<button class="button" (click)="enterTask(titleInput)">
  Add Task
</button>
```

This template consists of an input field as well as a button to enter a new task. Here, we're making use of the so-called local view variables by specifying that our input field should have the reference name `#titleInput`. We can reference this variable within the current component view by the name `titleInput`.

In this case, we are actually using the variable to pass the input field DOM element to the `enterTask` function that we call on a click event on the `Add Task` button.

Let's take a look at the implementation of our `Component` class for entering a new task by using the following code in a newly-created `enter-task.js` file:

```
import {Component, Output, ViewEncapsulation, EventEmitter} from
'@angular/core';
import template from './enter-task.html!text';

@Component({
  selector: 'ngc-enter-task',
  host: { class: 'enter-task' },
  template,
  encapsulation: ViewEncapsulation.None
})
export class EnterTask {
  // Event emitter that gets fired once a task is entered.
  @Output() taskEntered = new EventEmitter();
  // This function will fire the taskEntered event emitter
  // and reset the task title input field.
  enterTask(titleInput) {
    this.taskEntered.next(titleInput.value);
    titleInput.value = '';
    titleInput.focus();
  }
}
```

For this component, we've chosen a design approach where we use a loose relation to our task list where the actual task will be created. Although this component is closely related to the task list, it's better to keep the components as loosely coupled as possible.

One of the simplest forms of inversion of control, a callback function or event listener is a great principle to establish loose coupling. In this component, we are using the `@Output` annotation to create an event emitter. The output properties need to be instance fields that hold an event emitter within the component. On the component's HTML element, we can then use event bindings to capture any events emitted. This gives us great flexibility that we can use to create a clean application design, where we glue components together through the binding within the view:

Usage	**Description**
`@Output() outputProp = new EventEmitter();`	When `outputProp.next()` is called, a custom event with the name `outputProp` will be emitted on the component. Angular will look for event bindings on the component's HTML element (where the component is used) and execute them: `<my-comp (output-prop)= "doSomething()">` Within the expressions in event bindings, you will always have access to a synthetic variable called `$event`. This variable is a reference to the data emitted by `EventEmitter`.
`@Output('out') outputProp = new EventEmitter();`	Use this way of declaring your output properties if you'd want to name your events differently from what your property name is. In this example, a custom event with the name `out` will be fired when `outputProp.next()` is called: `<my-comp (out)= "doSomething()">`

Okay, let's use this newly created component to add new tasks within our `task-list` component. First, let's modify the existing template of the `task-list` component. Open the file, `task-list.html`, in the `task-list` component folder. We need to add the `EnterTask` component to the template and also handle the custom event that we're going to emit, once a new task is entered within the component:

```
<ngc-enter-task (taskEntered)="addTask($event)">
</ngc-enter-task>
<ngc-task *ngFor="let task of tasks"
          [task]="task"></ngc-task>
```

Since the output property within the `enter-task` component is called `taskEntered`, we can

bind it with the event binding attribute, `(taskEntered)=""`, on the host element.

Within the event binding expression, we then call a function on our `task-list` component called `addTask`. Also, we use the synthetic variable `$event`, which contains the task title emitted from the `enter-task` component. Now, whenever we push the button in our `enter-task` component and an event gets emitted from the component, we catch the event in our event binding and handle it within the `task-list` component.

We also need to make some minor changes to the `task-list` component's JavaScript file. Let's open `task-list.js` and modify it with the following changes:

```
...
// The component for entering new tasks
import {EnterTask} from './enter-task/enter-task';

@Component({
  ...
  directives: [Task, EnterTask]
})
export class TaskList {
  ...
  // Function to add a task from the view
  addTask(title) {
    this.tasks.push({
      title, done: false
    });
  }
}
```

The only thing we changed within the `task-list` component module is its ability to declare the `EnterTask` component in the directives property so that the compiler recognizes our `enter-task` component correctly.

We have also added a function, `addTask`, which will add a new task to our task list with a title that is passed to the function. Now the circle is closed and our event from the `enter-task` component is routed to this function within the view of the `task-list` component.

You can now start live server from your project directory in order to test the newly added functionality using the `live-server` command.

Recap

We have added a new subcomponent of the task list that is responsible for providing the UI logic to add new tasks. In other words, we have covered the following topics:

1. We created a subcomponent that is loosely coupled using output properties and event emitters.
2. We learned about the `@Output` annotation and how to use it to create output properties.
3. We used event bindings to link the behavior together, from the view of a component.

Custom UI elements

The standard UI elements in the browser are great, but sometimes, modern web applications require more complex and intelligent input elements than the ones available within the browser.

We'll now create two specific custom UI elements that we'll use within our application going forward in order to provide a nice user experience:

- **Checkbox**: There's already a native checkbox input in the browser, but sometimes, it's hard to fit it into the visual design of an application. Native checkboxes are limited in their styling possibilities, and therefore, it's hard to make them look great. Sometimes, it's those minor details that make an application look appealing.
- **Toggle buttons**: This is a list of toggle buttons, where only one button can be toggled within the list. They could also be represented with a native radio button list. However, like with native checkboxes, radio buttons are sometimes not really the nicest visual solution to the problem. A list of toggle buttons that also represents a select-one-user input element is much more modern and provides the visual aspect that we are looking for. Besides, who does not like to push buttons?

Let's create our custom checkbox UI element first. As we'll probably come up with a few custom UI elements, first let's create a new subfolder called `ui` within the `lib` folder.

Within the `ui` folder, we now create a folder with the name `checkbox` for our checkbox component. Starting with the template of our new component, we now create a file with the name `checkbox.html` within the `checkbox` folder:

```
<input type="checkbox"
       [checked]="checked"
       (change)="onChecke
dChange($event.target.checked)">
  {{label}}
```

On the `checkbox` input, we have two bindings. First, we have a property binding for the `checked` property on the DOM element. We are binding the DOM property to the `checked` member field on our component, which we are going to create in a moment.

Also, we have an event binding on the input element where we listen for the checkbox change DOM event and call the method `onCheckedChange` on our component class. We use the synthetic variable `$event` to pass the `checked` property on the checkbox DOM element where the change event is originated.

Moving on to our `component` class implementation, we need to create a file with the name `checkbox.js` within the `checkbox` folder:

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

```
import template from './checkbox.html!text';

@Component({
  selector: 'ngc-checkbox',
  host: { class: 'checkbox' },
  template,
  encapsulation: ViewEncapsulation.None
})
export class Checkbox {
  // An optional label can be set for the checkbox
  @Input() label;
  // If the checkbox is checked or unchecked
  @Input() checked;
  // Event emitter when checked is changed using the convention
  // for two way binding with [(checked)] syntax.
  @Output() checkedChange = new EventEmitter();

  // This function will trigger the checked event emitter
  onCheckedChange(checked) {
    this.checkedChange.next(checked);
  }
}
```

There's nothing special about this `component` class if we first look at it. It uses an input property to set the checked state from the outside, and it also has an output property with an event emitter that allows us to notify the outer component about the changes of the checked state using a custom event. However, there's a naming convention that makes this component a bit special. The convention of using an input property name also as an output property name but appending the word *change* is actually enabling a developer who uses the component to make use of the two-way data binding template shorthand.

Angular does not come with two-way data binding out of the box. However, creating two-way binding is quite easy. Actually, two-way data binding is no different than combining a property binding with an event binding.

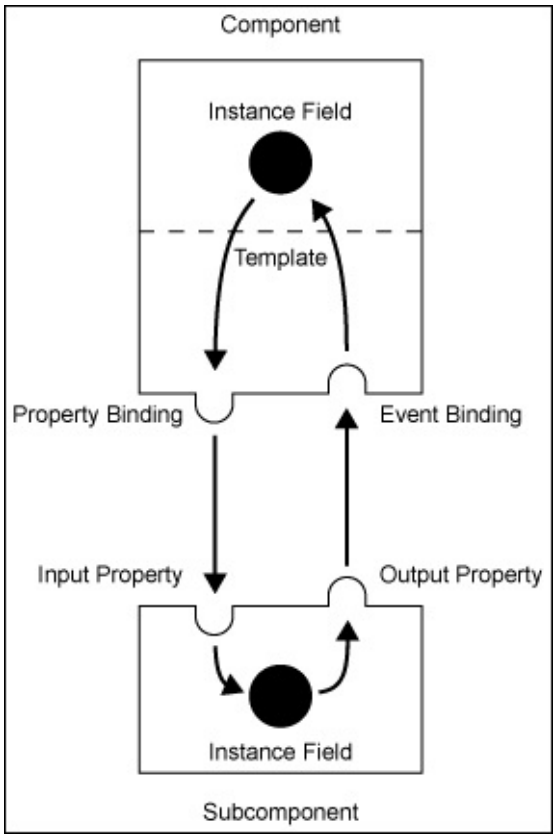
The following example creates a very simple two-way data binding process on an input field:

```
  <input type="text" (input)="value = $event.target.value"
                     [value]="value">
```

The simplicity of Angular and the general approach of extending the native functionality of the browser makes implementing this mechanism a breeze.

Implementing two-way data binding between a component and its subcomponent isn't really too difficult. The only thing we need to take care about is that there are input and output properties of the subcomponent involved.

Please have a look at the following screenshot:



A two-way data binding between member variables of a component and a subcomponent

Since two-way data binding was a highly requested feature in Angular, there's a handy shorthand to write it. Let's look at some examples on how to implement data bindings between a template of a component and its subcomponent:

Subcomponent properties	**Bindings in component template**

`@Input() text;`
 	
`@Output() textOut = new EventEmitter();`
	`<sc [text]="myText"`
	`(textOut)="myText = $event">`
	
	We bind the component's `myText` property to the subcomponent's text input. Also, we capture the `textOut` event emitted from the subcomponent and update our `myText` property.

`@Input() text;`
 	
`@Output() textChange = new`
	`<sc [(text)]="myText">`
	
	We can simplify this two-way data binding by using the naming convention to append the word "change" to our event emitter identifier. This way, we can use the two-way data binding shorthand within our

`EventEmitter();`	template using the `[(property)]` notation.

If we look at our `checkbox` component implementation again, we will see that we are using the two-way data binding naming convention for the checked property of our component. This way, we enable the use of the template shorthand for two-way data binding wherever we use our custom checkbox UI component.

Let's integrate our checkbox in the task component to replace the native checkbox input we're currently using there. For this, we need to modify the `task.html` file within the `task-list/task` folder, by replacing the native input checkbox that we have in the `task.html` file with the following line of code:

```
<ngc-checkbox [(checked)]="task.done"></ngc-checkbox>
```

As always, we also need to tell the task component that we'd like to use the component within the template. Let's change the code within the `task.js` file accordingly:

```
...
import {..., HostBinding} from '@angular/core';
// Each task has a checkbox component for marking tasks as done.
import {Checkbox} from '../../ui/checkbox/checkbox';

@Component({
  ...
  // We need to specify that this component relies on the Checkbox
  // component within the view.
  directives: [Checkbox]
})
export class Task {
  // Our task model can be attached on the host within the view
  @Input() task;

  @HostBinding('class.task--done')
  get done() {
    return this.task && this.task.done;
  }
}
```

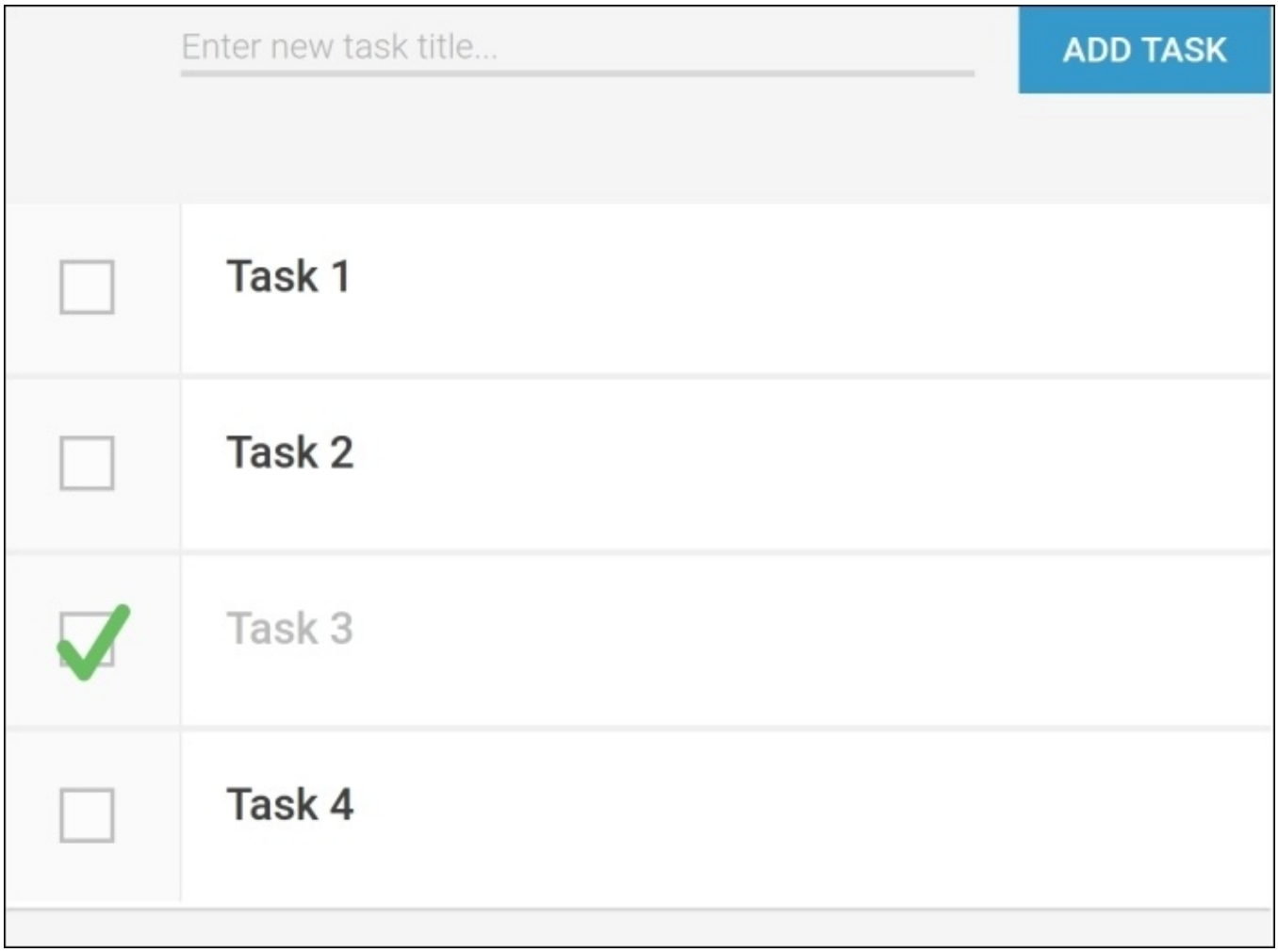
We've already learned about the host property on components. It allows us to set property and event bindings on our component host element. The host element is the DOM element where our component is initialized within the parent component.

There's another way through which we can set properties on our component host element, which becomes handy when we want to set a property based on some data within our component.

Using the `@HostBinding` annotation, we can create property bindings on the component host element based on the members within our component. Let's use this annotation in order to create a binding that will conditionally set the `task--done` class on the component's HTML

element. This is used to make some visual distinctions of finished tasks within our styles.

This was just the last step to integrate our custom checkbox UI component within the task component. You can now start `live-server` in order to view your changes and play around with these large new checkboxes in the task list. Isn't that much more fun to do than activating regular checkboxes? Don't underestimate the effect of a user interface that is pleasing to use. This can have a very positive impact on the usage of your product.



Our task list after adding our custom checkbox component

Now that we've created our `checkbox` component, let's go ahead and create another UI component for toggle buttons that we'll use in the next topic. We need to create a folder named `toggle` within the `ui` folder and create a template called `toggle.html` within the `toggle` folder:

```
<button class="button button--toggle"
        *ngFor="let button of buttonList"
        [class.button--active]="button === selectedButton"
        (click)="onButtonActivate(button)">{{button}}</button>
```

Nothing special here, really! We repeat a button by iterating over an instance field called

`buttonList` using the `NgFor` directive. This button list will contain the labels of our toggle buttons. Conditionally, we set a class called `button--active` using a property binding and checking it against our current button within the iteration against an instance field called `selectedButton`. When the button is clicked, we call a method, `onButtonActivate`, on our component class and pass the current button label from the iteration.

Let's create `toggle.js` inside the `toggle` folder and implement the `component` class:

```
import {Component, Input, Output, ViewEncapsulation, EventEmitter} from
'@angular/core';
import template from './toggle.html!text';

@Component({
  selector: 'ngc-toggle',
  host: {
    class: 'toggle'
  },
  template,
  encapsulation: ViewEncapsulation.None
})
export class Toggle {
  // A list of objects that will be used as button values.
  @Input() buttonList;
  // Input and state of which button is selected needs to refer to
  // an object within buttonList
  @Input() selectedButton;
  // Event emitter when selectedButton is changed using the
  // convention for two way binding with [(selected-button)]
  // syntax.
  @Output() selectedButtonChange = new EventEmitter();

  // Callback within the component lifecycle that will be called
  // after the constructor and inputs have been set.
  ngOnInit() {
    if (this.selectedButton === undefined) {
      this.selectedButton = this.buttonList[0];
    }
  }

  // Method to set selected button and trigger event emitter.
  onButtonActivate(button) {
    this.selectedButton = button;
    this.selectedButtonChange.next(button);
  }
}
```

Within our `toggle` component, we rely on the `buttonList` member to be an array of objects, as we are using this array within our template on an `NgFor` directive. The `buttonList` member is annotated to be an input property; this way, we can pass the array into the component.

For the `selectedButton` member, which holds the object of the `buttonList` array that is currently selected, we use a two-way data binding approach. This way, we can not only set the toggled button from the outside of the component, but also get notified via the `toggle`

component, when a button is toggled in the UI.

Within the `onButtonActivate` function, we are setting the `selectedButton` member as well as triggering the event emitter.

The `ngOnInit` method is actually called by Angular within the life cycle of directives and components. In the case where the `selectedButton` input property was not specified, we'll add a check and select the first button from the available button list. Since `selectedButton` as well as `buttonList` are instance fields that are also input properties at the same time, we need to wait for them to be initialized in order to execute this logic. It's important not to perform this initialization within the component constructor. The life cycle hook, `OnInit`, will be called after the directive input and output properties have been checked for the first time. It is invoked only once when the directive is constructed.

Tip

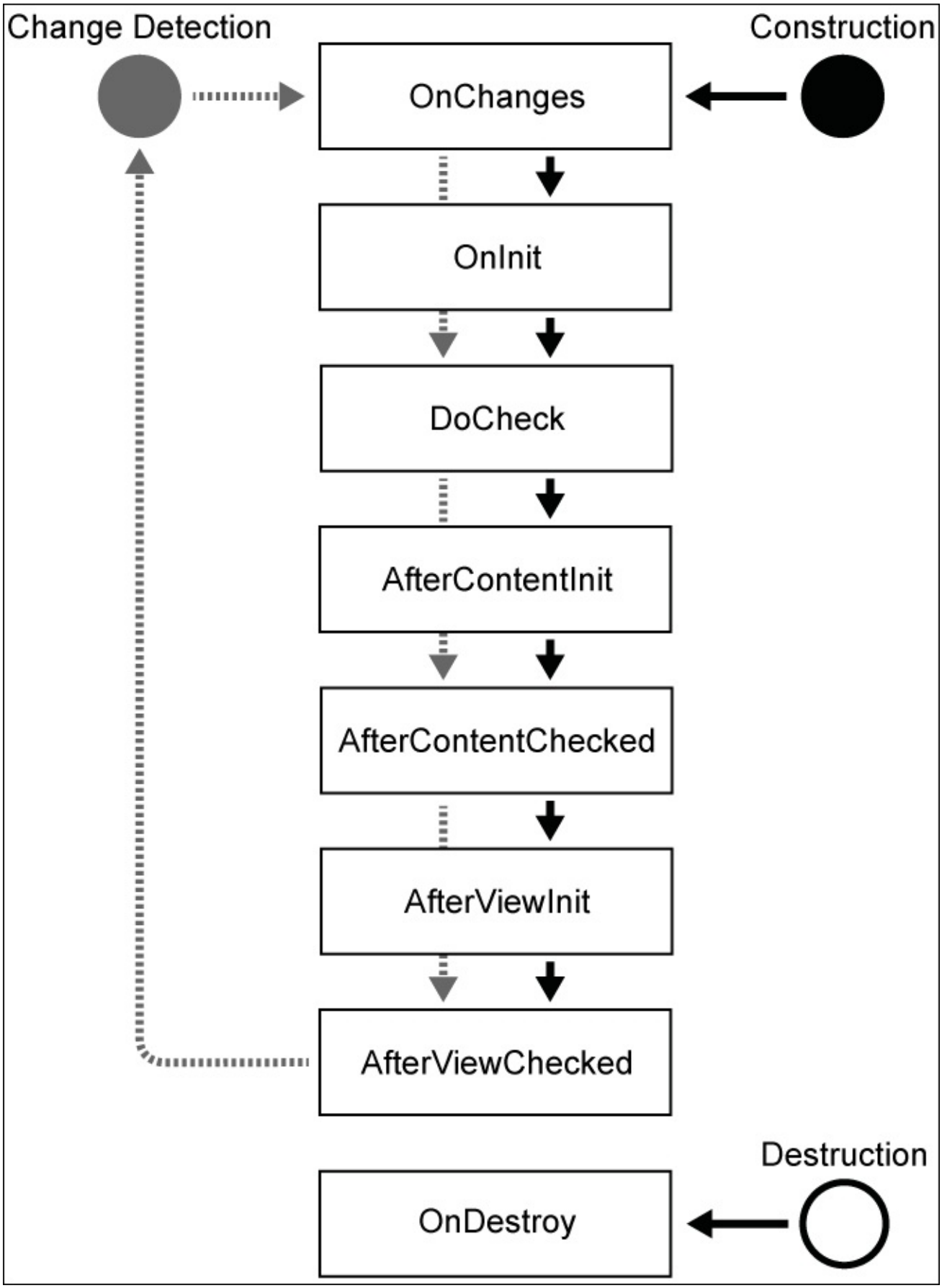
Angular will call any life cycle hooks that have been implemented on your component automatically.

The next diagram illustrates the life cycle of an Angular component. Upon component construction, all the life cycle hooks will be called as per the order shown in the diagram, except the `OnDestroy` hook, which will be called upon component destruction.

Change detection will also start a subset of life cycle hooks, where there will be at least two cycles in the following order:

- `doCheck`
- `afterContentChecked`
- `afterViewChecked`
- `onChanges` (if any changes are detected)

A detailed description of the life cycle hooks and their purpose is available on the Angular documentation website at https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html.



An illustration of the life cycle of an Angular component

Recap

In this block, you learned how to build custom UI components that are generic and loosely coupled so that they can be used in other components as subcomponents. We also completed the following tasks:

1. We created a subcomponent that is loosely coupled using output properties and event emitters.
2. We learned what the `@Output` annotation is and how to use it to create output properties.
3. We used the `@HostBinding` annotation to create property bindings declaratively from within our component class.
4. We used event bindings to link the behavior together from the view of a component.
5. We built two-way data binding using a binding shorthand.
6. We learned about the working of the Angular component life cycle and how we can use the `OnInit` life cycle hook to initialize the component after the input and output have been processed for the first time.

Filtering tasks

This is the last building block of this chapter. We have already learned a lot about building basic components and how to compose them together in order to form larger components. In the previous building block, we created generic UI components that could be used in other components. In this topic, we will use the toggle button component not only to create a filter for our task list, but also to improve the way we receive and store tasks by using data services.

Let's continue with another refactoring exercise. So far, we have stored our task list data directly within the `task-list` component, but let's change that here and use a service that will provide tasks for us.

Our service will still not use a database, but we'll get the task data out of our component. In order to use the service, we're making use of Angular's dependency injection for the first time.

Let's create a new file called `task-list-service.js` within the `lib/task-list` folder of our application:

```
// Classes which we'd like to provide for dependency injection
// need to be annotated using this decorator
import {Injectable} from '@angular/core';

@Injectable()
export class TaskListService {
  constructor() {
    this.tasks = [
      {title: 'Task 1', done: false},
      {title: 'Task 2', done: false},
      {title: 'Task 3', done: true},
      {title: 'Task 4', done: false}
    ];
  }
}
```

We've moved all our task data into the newly created service. In order to make our service class injectable, we need to decorate it with the `@Injectable` annotation.

Let's apply some changes to our `task-list` component and modify the `task-list.js` file within the `task-list` folder. The modified code in the file is highlighted in the following code excerpt:

```
import {..., Inject} from '@angular/core';
// The dummy task service where we get our tasks from
import {TaskListService} from './task-list-service';
...

// We also need a Toggle UI component to provide a filter
import {Toggle} from '../ui/toggle/toggle';
```

```
@Component({
  ...
  // Set the TaskListService as host provider
  providers: [TaskListService],
  // Specify all directives / components that are used in the view
  directives: [Task, EnterTask, Toggle]
})
export class TaskList {
  // Inject the TaskListService and set our filter data
  constructor(@Inject(TaskListService) taskListService) {
    this.taskListService = taskListService;
    this.taskFilterList = ['all', 'open', 'done'];
    this.selectedTaskFilter = 'all';
  }
  // Method that returns a filtered list of tasks based on the
  // selected task filter string.
  getFilteredTasks() {
    return this.taskListService.tasks ?
this.taskListService.tasks.filter((task) => {
      if (this.selectedTaskFilter === 'all') {
        return true;
      } else if (this.selectedTaskFilter === 'open') {
        return !task.done;
      } else {
        return task.done;
      }
    }) : [];
  }
  // Method to add a task from the view
  addTask(title) {
    this.taskListService.tasks.push({
      title,
      done: false
    });
  }
}
```

In the import section of our module, we're going to import the task list service. We will use dependency injection to receive an instance of the `TaskListService` class within our component constructor. For this, we'll use a new annotation, which lets us specify the type we'd like to inject. The `Inject` decorator needs to be imported from the Angular core module in order to use the `@Inject` annotation. If you take a look at our constructor, you'll find that we're using the `@Inject` annotation there to specify what instance type we'd like to inject.

In addition to the `@Inject` annotation on the constructor, we need one last thing to make the injection work. We need to register `TaskListService` as a provider within the `providers` property of our `@Component` annotation.

Now we get the `TaskListService` injected when the directive is constructed, and we can store a reference to it inside an instance field.

Within the constructor of the component, we also want to store a list of states the task status

filter can have. This list will also serve as input for our toggle button list. If you recall the input properties on our toggle button, we have a `buttonList` input that accepts a list of button labels. To store the currently selected filter type, we use an instance field called `selectedTaskFilter`.

The last piece that we need to add to our `task-list` component is the method, `getFilteredTasks`. We no longer need to store the task list directly within an instance field, and tasks should only be received within the component using this method. The logic inside the method checks the `selectedTaskFilter` property and returns a filtered list that meets this condition.

Since we want to use the toggle button component that we've created within the previous topic to create a filter button list, we will need to import the toggle component within the import section and also add the `Toggle` class to our `directives` property. Now we can use the toggle component within the template of our `task-list` component.

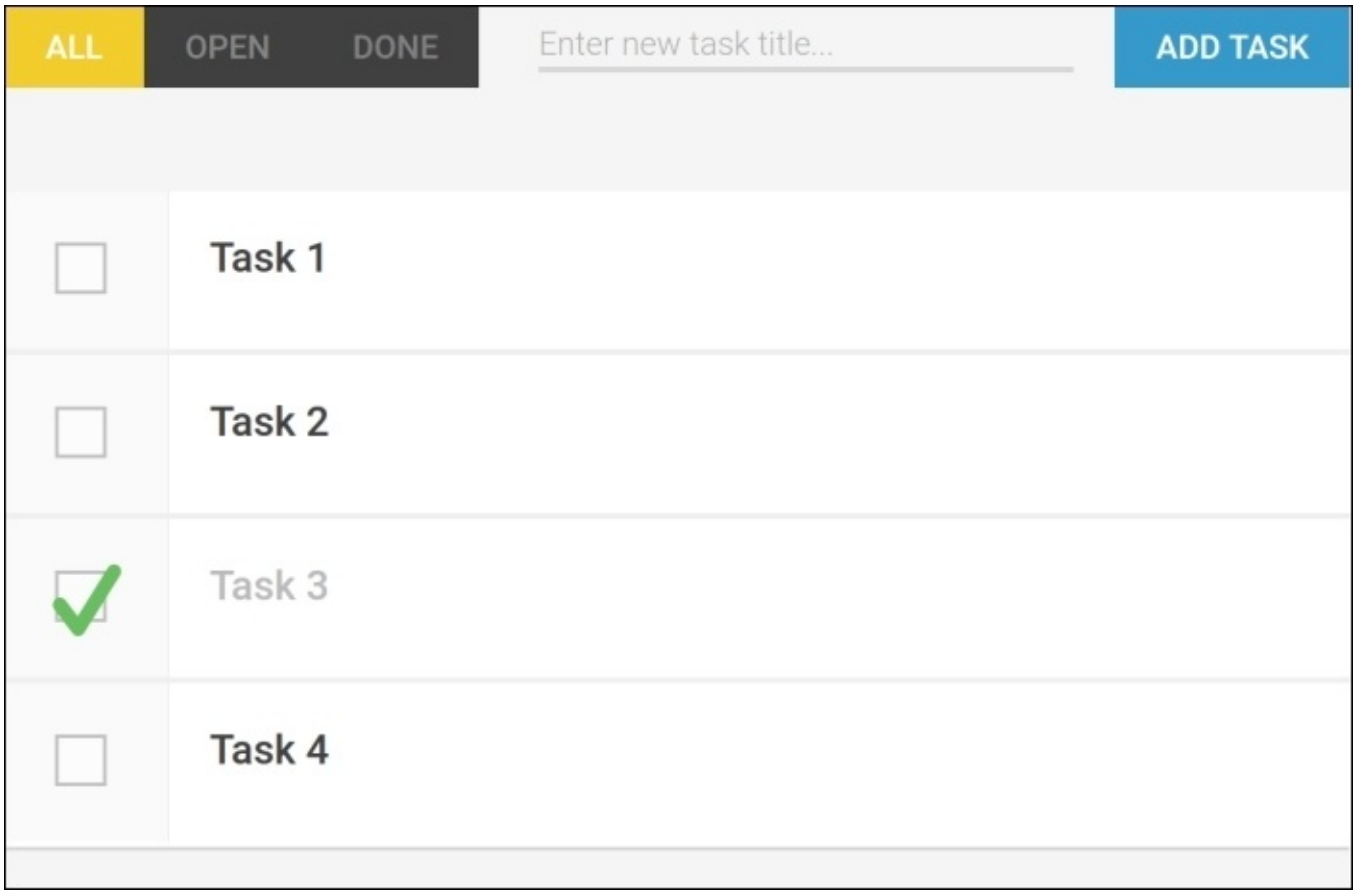
Okay, that's all we are going to change in our component implementation. We want to change our view template though to use the filtered task list coming from the data service and show a toggle button list to activate the different filter types. Let's open the template file, `task-list.html`, inside the `task-list` folder and modify it with the following content:

```
<ngc-toggle [buttonList]="taskFilterList"
            [(selectedButton)]="selectedTaskFilter">
</ngc-toggle>
<ngc-enter-task (taskEntered)="addTask($event)">
</ngc-enter-task>
<ngc-task *ngFor="let task of getFilteredTasks()"
          [task]="task"></ngc-task>
```

Since we've added the toggle component within the `directives` property of our `task-list` component, we can use it now within our view template. We bind the input property `buttonList` to `taskFilterList` that we store within our `task-list` component. Also, we're using two-way data binding to bind the `selectedButton` input property of the toggle button list to the `selectedTaskFilter` instance field of the task list. This way, we can not only update the selected task filer from our `task-list` component programmatically, but also allow a user to change the value using the toggle button list.

Now we only need to make a small change to the `NgFor` directive that repeats our task elements within the task list. Since we need to access the tasks of the `task-list` component with the `getFilteredTasks` method now, we also need to use that method within our repeater expression.

That's it already, congratulations! You've successfully added a filtering mechanism to your task list by reusing the toggle component that we created in the previous topic. You can now start your live server (using the `live-server` command) and should see a fully functional task list where you can enter new tasks and also filter the task list:



Screenshot of the task list with the newly added toggle button component for filtering the task state

Summary

In this chapter, you learned a lot of new concepts on building UI-component-based applications with Angular. Also, we built the core component of our task management application, which is the task list itself. You learned about the concept of input and output properties and how to use them to build two-way data binding.

We also covered the basics of the Angular component life cycle and how to use life cycle hooks to execute post initialization steps.

As the last step, we integrated a toggle button list component within our task list to filter the task states. We refactored our `task-list` component to use a service in order to obtain task data. For this, we used Angular's dependency injection.

Chapter 3. Composing with Components

In this chapter, we will go one step further in structuring our application and working on the layout and architecture that serves as the base for our task management system. Besides introducing new components and creating larger compositions with the existing components, we'll also look at the way that we deal with data. So far, we've obtained task data synchronously from the `TaskListService` that we created in the previous chapter. However, in real-world scenarios, this would rarely be the case. In a real application, data is mostly retrieved in an asynchronous form. Usually, we acquire data through a RESTful web service, and we use `XMLHttpRequest` or the recently standardized fetch API. However, as we're trying to build a cutting-edge application, we will go one step further. In this chapter, we'll look at how we can restructure our application to deal with observable data structures using RxJS—a functional and reactive programming library that is used in Angular.

In this chapter, we will look at the following topics:

- Restructuring our application to deal with observable data structures
- The basics of RxJS and its operators in order to build a reactive data model
- Using pure components in Angular
- Using `ChangeDetectionStrategy.OnPush` for pure components
- Using content projection points and `@ContentChildren` to create a `Tab` component
- Creating a simple navigation component
- Injecting parent components and establishing direct component communication
- Combining internal and external content to create a flexible component API

Data – Fake to real

Starting with this chapter, we are switching to a document-based database to store our tasks and project data. As a data store, we use the PouchDB project, which is an in-browser database that is designed to run with IndexedDB and various fallback strategies. PouchDB is designed similarly to Apache CouchDB, and it can even be synchronized with it.

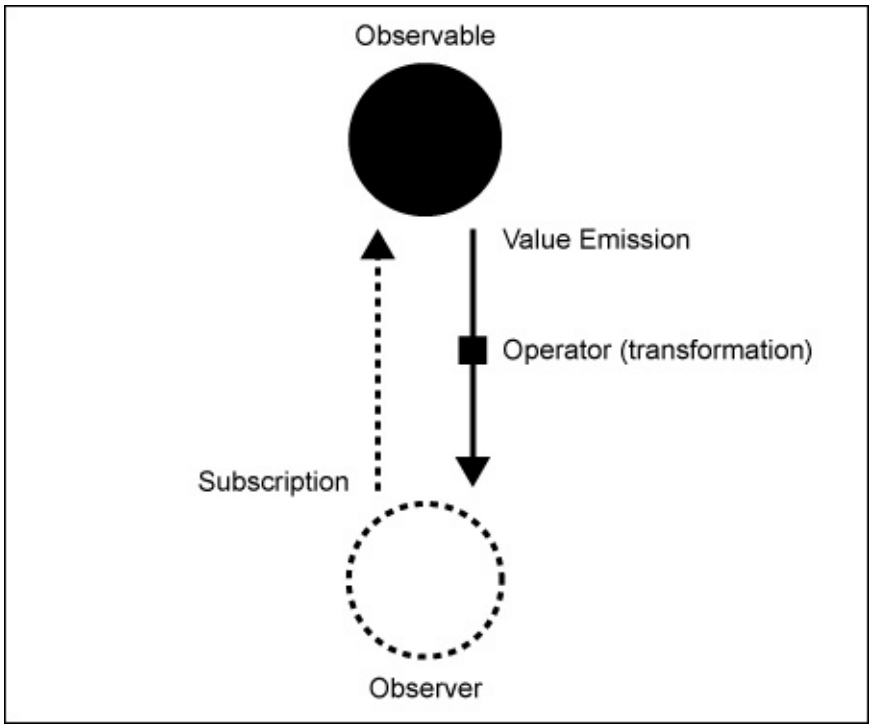
In order to provide a quality experience for you while you build your application, it's important that we work in real-life conditions. This means that we should use asynchronous data in our components and not rely on a simple JavaScript array of data. In order to make this as smooth as possible, the whole data layer is already set up for you, and you don't need to worry about the internals too much. Of course if you're still interested, I'm not holding you back from exploring the source code that is in the `data-access` folder.

Reactive programming with observable data structures

So far, we used simple array data structures in the task list that we created. This is not really what we'll find in real-world scenarios. In real applications, we have to deal with asynchronous data and the changes of the data that needs to be synchronized between users. The requirements for modern applications sometimes even go further and also provide view updates on the changed data in real time. As we're building a modern task management system here, we should try to keep up with these requirements.

Both of these, handling asynchronous data and handling real-time data updates, require a major redesign of the data flow in our application. Using observable data structures, we enable our application to master the challenges of asynchronous data where we need to react to change.

Handling data in applications behaves very similarly to streams. You take input, transform it, combine it, merge it, and finally, write it into output. In systems such as this, it's also very likely that input is in a continuous form and sometimes even of infinite duration. Just take a live feed as an example; this type of data flows continuously, and the data also flows infinitely. Functional and reactive programming are paradigms to help us deal with this kind of data in a cleaner way.



A simple observable subscription with value emission and a transformation

Angular 2 is reactive at its very core and the whole of the change detection and bindings are built using a reactive architecture. The input and output of components, which we've learned about in the previous chapter, is nothing but a data flow that is established using a reactive event-driven approach. Angular uses RxJS, a functional and reactive programming library for JavaScript, to implement this data flow. In fact, the `EventEmitter`, which we've used to send custom events from within our components, is just a wrapper around an RxJS observable.

Reactive and functional programming is exactly what we are looking for to redesign our application in order to handle asynchronous data and data changes. As we already have RxJS at hand from the production dependency of Angular, let's use it to establish a continuous data flow from our data source into our application. The `DataProvider` service that is present in the `data-access` folder of our project provides a nice wrapper around our data store using RxJS. As we will use this service in our whole application, we can directly provide it to the bootstrap in the `bootstrap.js` file, as follows:

```
// Import Angular bootstrap function
import {bootstrap} from '@angular/platform-browser-dynamic';
import {DataProvider} from '../data-access/data-provider';
// Import our main app component
import {App} from './app';

bootstrap(App, [
  DataProvider
]);
```

As a second argument to the `bootstrap` function of Angular, we can provide application-level dependencies, which will be available for injection in all components and directives.

Let's now use the `DataProvider` service as an abstraction to obtain data from the PouchDB data store and create a new service responsible to provide project data.

We will create a new `ProjectService` class on the `lib/project/project-service/project-service.js` path, as follows:

```
import {Injectable, Inject} from '@angular/core';
import {ReplaySubject} from 'rxjs/Rx';
import {DataProvider} from '../../../data-access/data-provider';

@Injectable()
export class ProjectService {
  constructor(@Inject(DataProvider) dataProvider) {
    …
  }
}
```

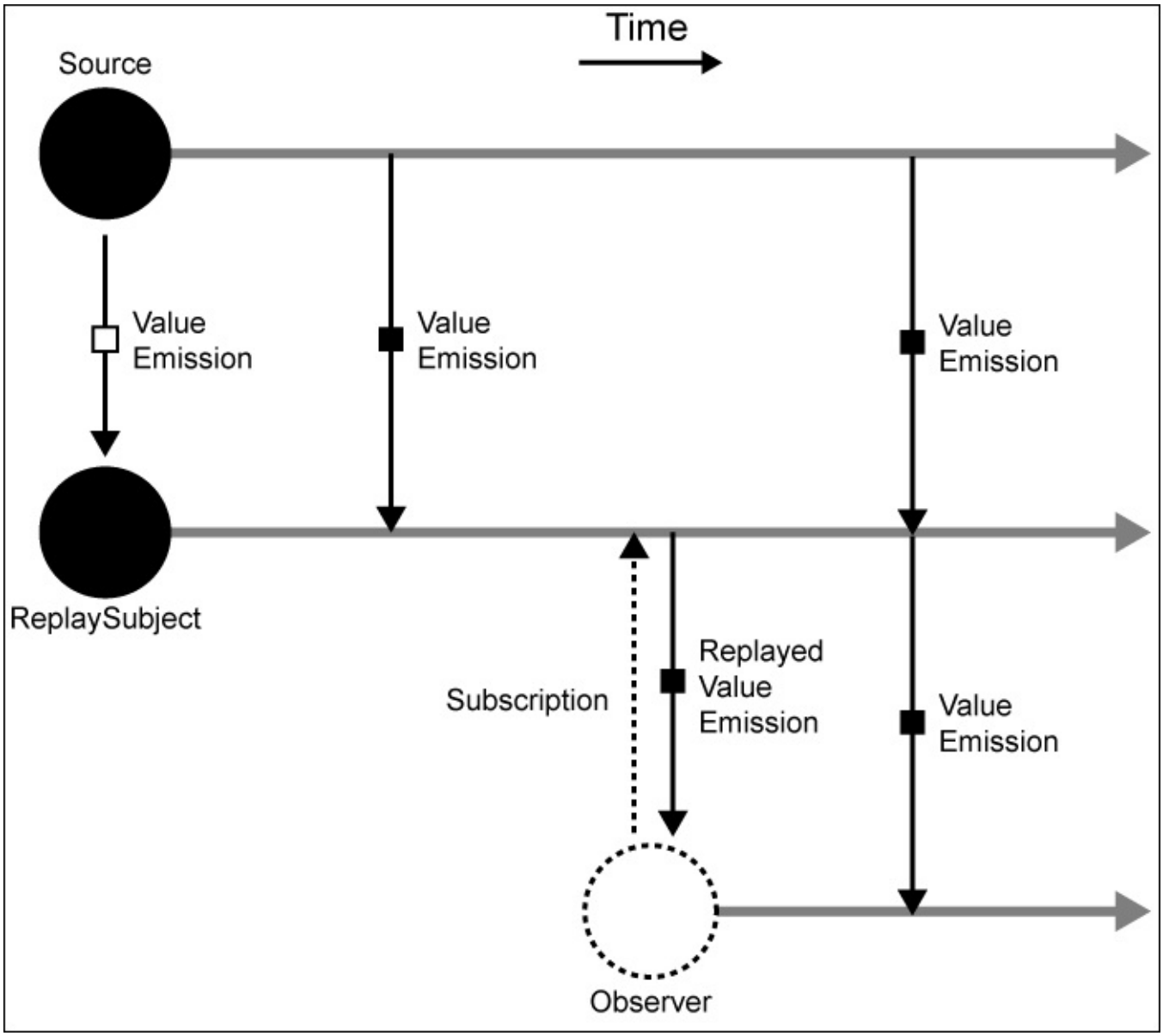
Looking at the import section of our new module, you can see that we import the necessary dependencies from the Angular core module for dependency injection. Our service class uses the `@Injectable` decorator so that we can provide this to the injectors of components. We also

inject the `DataProvider` service into the constructor of our newly-created service.

The `ReplaySubject` class, which we import from the RxJS library, is used to make our service reactive. A subject in the RxJS world is both an observer as well as an observable. It can observe something for changes and then emit further on to all its subscribers. You can think of a subject like a proxy, where it sits in the middle between a source for changes and a group of observers. Whenever the source emits changes, the subject will notify all subscribers about these changes.

Now, the `ReplaySubject` class is a special kind of subject that allows you to replay a buffer of changes when new subscribers get added. This is especially useful if you always need to provide some initial data to subscribers. Imagine our data, which we'd like to get propagated into the UI. We want to immediately get the initial data when we subscribe to our service and then going forward, we also want to get notified about changes. Using a `ReplySubject` class, which is buffering just one change, suits this use-case perfectly.

Let's look at the following figure, which illustrates the behavior of `ReplaySubject`:



A source connected to an observer using a ReplaySubject class, which buffers the most recent value and emits on subscription

In the preceding figure, you can see that we're connecting a `ReplaySubject` class to a source that is emitting value changes over time. After two emissions, an observer subscribes to our `ReplaySubject` class. `ReplaySubject` will then replay all buffered changes to the new subscriber as if these events just occurred. In this example, we use a replay buffer length of one. On subsequent value emissions, these will be directly re-emitted to the subscribers of the `ReplaySubject` class.

Let's go back to our `ProjectService` class and add some logic to the `constructor` function in order to emit project data using a `ReplaySubject` class.

We will start off with some member field initialization, for which we're going to need to

implement the logic:

```
this.dataProvider = dataProvider;
this.projects = [];
// We're exposing a replay subject that will emit events whenever
// the projects list change
this.change = new ReplaySubject(1);
```

Note that we created a new `ReplaySubject` class with a buffer length of one, and assign it to the member filed with the name `change`.

We also assign the `DataProvider` service, which was previously injected in the constructor parameters, to the `dataProvider` member field.

Now, it's time to make use of the `DataProvider` service in order to subscribe for any changes within our data store. This establishes a reactive connection to our data that is stored in PouchDB:

```
// Setting up our functional reactive subscription to receive
// project changes from the database
this.projectsSubscription = this.dataProvider.getLiveChanges()
  // First convert the change records to actual documents
  .map((change) => change.doc)
  // Filter so that we only receive project documents
  .filter((document) => document.type === 'project')
  // Finally we subscribe to the change observer and deal with
  // project changes in the function parameter
  .subscribe((changedProject) => {
    this.projects = this.projects.slice();
    // On every project change we need to update our projects list
    const projectIndex = this.projects.findIndex(
      (project) => project._id === changedProject._id
    );
    if (projectIndex === -1) {
      this.projects.push(changedProject);
    } else {
      this.projects.splice(projectIndex, 1, changedProject);
    }
    // Emit an event on our replay subject
    this.change.next(this.projects);
  });
```

The observable that is returned by the `getLiveChanges()` function emits data in our data store as changes. In addition to this, this will also emit any future changes that are applied to our store after we've received the initial data. You can imagine a persistent connection to the database, and whenever a document is updated in the database, our observer will receive this change as a value.

Observables provide a large amount of so-called operators that allow you to transform the data stream that originated at the observable. You might already know about some of these functional operators from the ECMAScript 5 array extra functions, such as `map` and `filter`.

Using operators, you can model a whole transformation flow until you finally subscribe to the data.

As we receive changed objects from the data store, we first need to transform them into document objects. This is fairly easy because each change object contains a `doc` property that actually holds the whole data of the changed document for which we have received an update. Using the `map` function, we can transform the changed objects into project objects before we return them back into the data flow:

```
.map((change) => change.doc)
```

`DataProvider` will provide us with data for all the documents in the store. As we are only interested in project data at the moment, we also apply a filter that filters out all the documents that are not of the `project` type:

```
.filter((document) => document.type === 'project')
```

Finally, we can subscribe to the transformed stream, as follows:

```
.subscribe((changedProject) => {})
```

The `subscribe` operator is where we terminate our observation path. This is like an endpoint at which we sit and observe. Inside our `subscribe` function, we listen for project document updates and incorporate them into the `projects` member property of our `App` component. This includes not only adding the new projects that were added to the document store, but it also includes updating existing projects. Each project contains a unique identifier that can be accessed by the `_id` property. This allows us to find the right action easily.

After updating our actual view of the projects and storing the list of projects in the `projects` member field, we can emit the updated list using our `ReplaySubject` class:

```
this.change.next(this.projects);
```

Our `ProjectService` class is now ready to be used, and applications components that need to obtain project data can subscribe to the exposed `ReplaySubject` class in order to react on these data changes.

Let's refactor our `App` component in `lib/app.js` and get rid of the fake `TaskListService` that we've used so far:

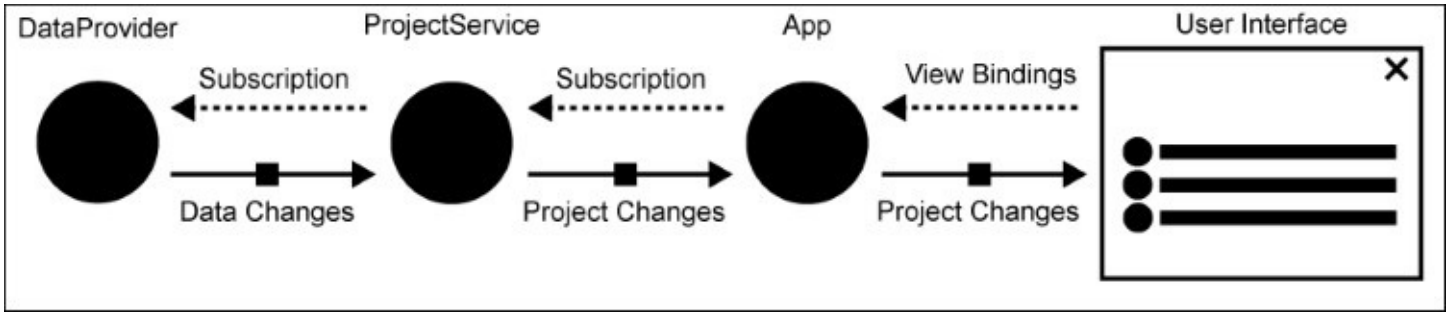
```
import {Component, ViewEncapsulation, Inject} from '@angular/core';
import {ProjectService} from './project/project-service/project-service';
import template from './app.html!text';

@Component({
  selector: 'ngc-app',
  template,
  encapsulation: ViewEncapsulation.None,
  providers: [ProjectService]
})
```

```
export class App {
  constructor(@Inject(ProjectService) projectService) {
    this.projectService = projectService;
    this.projects = [];

    // Setting up our functional reactive subscription to receive
    // project changes
    this.projectsSubscription = projectService.change
      // We subscribe to the change observer of our service
      .subscribe((projects) => {
        this.projects = projects;
      });
  }

  // If this component gets destroyed, we need to remember to
  // clean up the project subscription
  ngOnDestroy() {
    this.projectsSubscription.unsubscribe();
  }
}
```

In our `App` component, we now obtain a list of projects using the changed observable on our `ProjectService` class. Using a reactive subscription to `ReplaySubject` on the service, we make sure that our projects list that is stored in the `App` component will always be updated after any changes.

We use the `OnDestroy` lifecycle hook to unsubscribe from the `ProjectService` change observable. This is a necessary manual step if you like to do proper housekeeping in your application. Depending on the source, forgetting to unsubscribe could lead to memory leaks.

With the preceding code, we already established the base for a reactive data architecture. We observe our data store for changes and our user interface will react on them:



This figure shows the reactive data flow end-to-end from our data-store into the view

Immutability

Immutable data was originally a core concept of functional programming. This topic will not cover immutable data in much depth, but it will explain the core concept so that we can talk about how to apply this concept to Angular components.

Immutable data structures force you to create a full copy of the data that you want to modify before you can do this. You'll never operate on the data directly but on a copy of this same data. This has many benefits over standard mutable data, the most obvious probably being the application state. When you always operate on new copies of data, there's no chance that you're messing up the data that you actually didn't want to modify.

Let's take this simple example, which illustrates the issues object references can cause:

```
const list = [1, 2, 3];
console.log(list === list.reverse()); // true
```

Although this seems odd at first, it actually makes sense that the output of this example is true. `Array.reverse()` is a mutable operation, and it will modify the innards of the array. The actual reference will stay the same because JavaScript will not create a copy of the array to reverse it. Although technically this makes a lot of sense, this is not what we expected in the first place when we look at this code.

We can easily change this example to an immutable procedure by creating a copy of the array before we reverse it:

```
const list = [1, 2, 3];
console.log(list === list.slice().reverse()); // false
```

The issue with references is that they can cause a lot of unexpected side-effects. Also, if we come back to our encapsulation topic from Chapter 1, *Component-Based User Interfaces*, they are completely against the concept of encapsulation. Although we might think that it would be safe to pass complex data types into a capsule, it's actually not. As we're dealing with references here, the data can still be modified from the outside, and our capsule will not have the complete ownership. Consider the following example:

```
class Sum {
  constructor(data) {
    this.data = data;
    this.data.sum = data.a + data.b;
  }
  getSum() {
    return this.data.sum;
  }
}

const data = {a: 5, b: 8};
var sum = new Sum(data);
console.log(sum.getSum()); // 13
```

```
console.log(data.sum); // 13
```

Even if we only wanted to store the data internally in our `Sum` class, we would have created an unwanted side-effect of referencing and modifying the data object that's outside the instance. Multiple `sum` instances would also share the same data from outside and cause more side-effects. As a developer, you've learned to treat object references right, but they still can cause a lot of problems.

We don't have these problems with immutable data, which can be illustrated easily with primitive data types in JavaScript. Primitive data types don't use references, and they are immutable by design:

```
let originalString = 'Hello there!';
let modifiedString = originalString.replace(/e/g, 3);
console.log(originalString); // Hello there!
console.log(modifiedString); // H3llo th3r3!
```

There's no way we can modify an instance of a string. Every modification that we perform on a string will generate a new string, and this prevents the unwanted side-effects.

So, why do we still have object references within programming languages, even though they cause so many issues? Why aren't we performing all these operations on immutable data, and why are we only dealing with values rather than object references?

Of course, imperative data structures also come with their benefits, and it always depends on the context if immutable data brings value.

One of the main reasons that is often used against immutable data is bad performance. Of course, it costs some performance if we need to create tons of copies of our data every time we want to modify it. However, there are great optimization techniques, which fully eliminate the performance issues that we would usually expect from immutable data structures. Using a tree data structure that allows internal structural sharing, copies of the data will be shared internally. This allows very efficient memory management, which in some situations, even outperforms mutable data structures. I can highly recommend the paper by Chris Okasaki about *Purely Functional Data Structures* if you would like to read more about performance in immutable data structures.

Tip

JavaScript does not support immutable data structures out of the box. However, you can use libraries, such as `Immutable.js` by Facebook, which provide you with an easy API to deal with immutable data. `Immutable.js` even implements structural sharing and makes it a perfect power tool if you decide to build on an immutable architecture in your application.

As with every paradigm, there are pros and cons, and depending on the context, one concept may fit better than another one. In our application, we won't use immutable data structures that are provided by third-party libraries, but we'll borrow some of the benefits that you get from

immutable data by the following immutable idioms:

- **It's much easier to reason about immutable data**: You can always tell why your data is in a given state because you know the exact transformation path. This may sound irrelevant, but in practice, this is a huge benefit not only for humans to write code, but also for compilers and interpreters to optimize it.
- **Using immutable objects makes change detection much faster**: If we rely on immutable patterns to treat our data, we can rely on object reference checks to detect change. We no longer need to perform complex data analysis and comparison for dirty checking, and can fully rely on checking references. We have the guarantee that object properties don't change without the object identity changing as well. This makes change detection as easy as `oldObject === newObject`.

Pure components

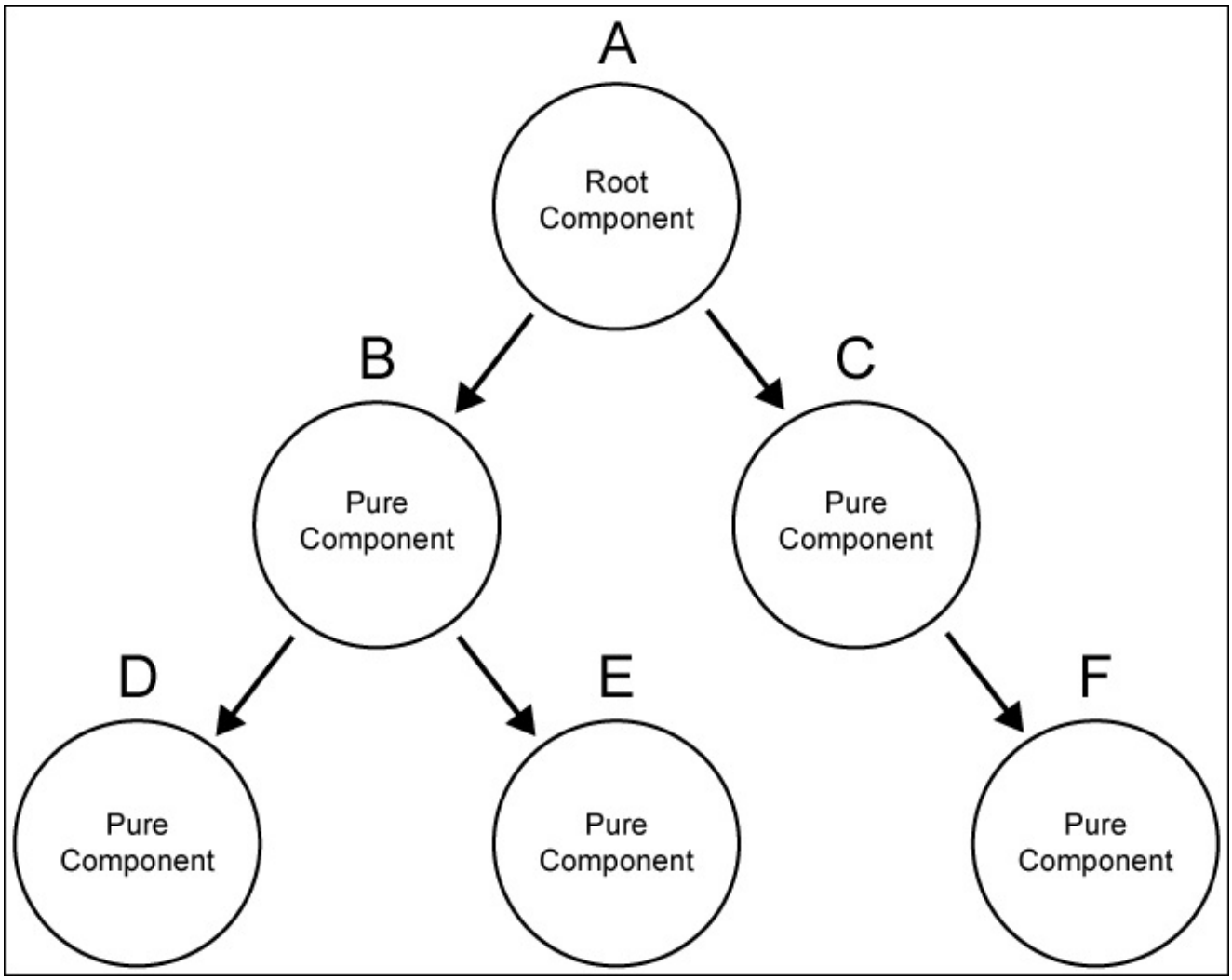
The idea of a "pure" component is that its whole state is represented by its inputs, where all inputs are immutable. This is effectively a stateless component, but additionally, all the inputs are immutable.

I like to call such components "pure" because their behavior can be compared to the concept of pure functions in functional programming. A pure function is a function which has the following properties:

- It does not rely on any state outside of the function scope
- It always behave the same if input parameters don't change
- It never changes any state outside the function scope (side-effect)

With pure components, we have a simple guarantee. A pure component will never change without its input parameters being changed. We can ignore a component and its subcomponents in change detection until one of the component inputs changes. Sticking to this idea about components gives us several advantages.

It's very easy to reason about pure components and their behavior can be predicted very easily. Let's look at a simple illustration of a component tree where we use pure components:



A component tree with immutable components

If we have the guarantee that each component in our tree has a stable state until an immutable input property changes, we can safely ignore change detection that would usually be triggered by Angular. The only way that such a component could change is if an input of the component changes. Let's say that there's an event that causes the **A** root component to change the input binding value of the **B** component, which will change the value of a binding on the **E** component. This event, and the resulting procedure, would mark a certain path in our component tree to be checked by change detection:

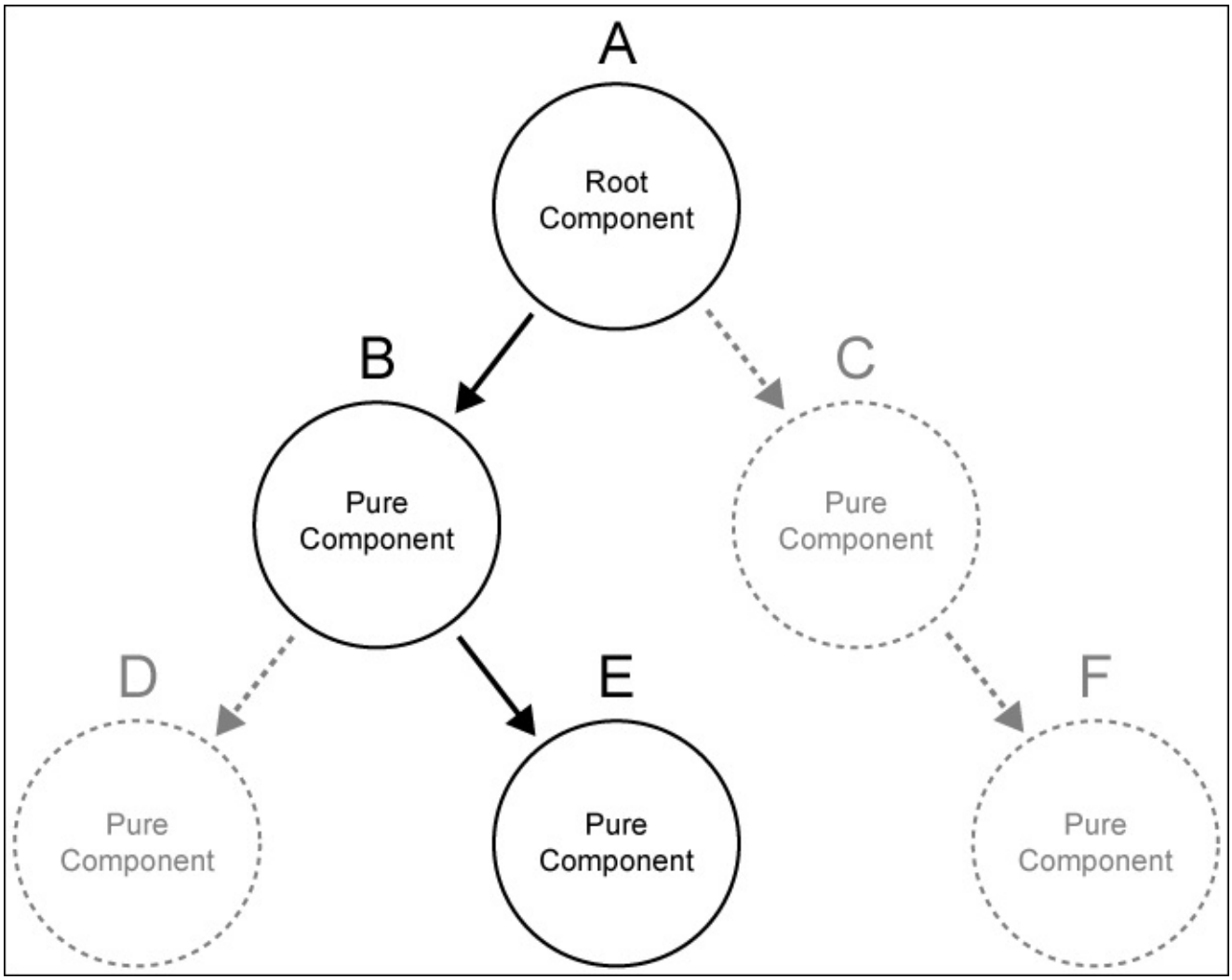


Figure that shows a marked path for change detection (in black) with "pure" components.

Although the state of the root component changed, which also changed input properties of the subcomponents on two levels, we only need to be concerned about a given path when thinking about possible changes in the system. Pure components give us the promise that they will not change if their inputs will not. Immutability plays a big role here. Imagine that you're binding a mutable object to the component, **B**, and the **A** component would change a property of this object. As we use object references and mutable objects, the property would also be changed for the **B** component. However, there's no way for the **B** component to notice this change, as we can't track who knows about our object within the component tree. Basically, we'd need to go back to regular dirty checking of the whole tree again.

By knowing that all our components are pure and that their inputs are immutable, we can tell Angular to disable change detection until an input property value changes. This makes our component tree very efficient, and Angular can optimize change detection effectively. When thinking about large component trees, this can make the difference between a stunningly-fast application and a slow one.

The change detection of Angular is very flexible, and each component gets its own change detector. We can configure the change detection of a component by specifying the `changeDetection` property of the component decorator.

Using `ChangeDetectionStrategy`, we can choose from a list of strategies that apply for the change detection of our component. In order to tell Angular that our component should only be checked if an immutable input was changed, we can use the `OnPush` strategy, which is designed exactly for this purpose.

Let's take a look at the different configuration possibilities of component change-detection strategies and some possible use cases:

Change-detection strategy	**Description**
`CheckAlways`	This strategy tells Angular to check this component during every change-detection cycle, and this is obviously the most expensive strategy. This is the only strategy that guarantees that a component gets checked for changes on every possible application state change. If we're not working with stateless or immutable components, or we are using an inconsistent data flow within our application, this is still the most reliable change-detection method. Change detection will be executed on every browser event that runs within the zone of this component.
`Detached`	This strategy tells Angular to completely detach a component subtree from change detection. This strategy can be used to create a manual change-detection mechanism.
`OnPush`	This strategy tells Angular that a given component subtree will only change under one of the following conditions: • One of the input properties changes where changes need to be immutable • An event binding within the component subtree is receiving an event
`Default`	This strategy simply evaluates to `CheckAlways`

Purifying our task list

In the previous topic, we changed our main application component to use RxJS Observables in order to get notified about the data changes in our data store.

We also looked into the basics of using immutable data structures and that Angular can be configured to assume component changes only occur when component input changes ("pure" components). As we'd like to get the performance benefits that result from this optimization, let's refactor our task list component to make use of this.

In the previous chapter, we built our `TaskList` component and directly placed the task data in the component. We then refactored our code so that we could place the task data into a service and use injection to obtain data.

Now, we're rebuilding our `TaskList` component to make it "pure" and only dependent on its input properties. As we'll make sure that the data flowing into the component is always immutable, we can use the `OnPush` change-detection strategy on our refactored component. This will certainly give our task list a performance boost.

Equally important as performance, are the structural benefits that we get from using pure components. A "pure" component does not change any data directly because it's not allowed to modify application state. Instead, it uses output properties to emit events on changed data. This allows our parent component to react to these events and perform the necessary steps to handle the changes. As a result of this, the parent component will possibly change the input properties of the pure component. This will trigger change detection and effectively change the state of the pure component.

What might sound a bit overcomplicated at first is actually an immense benefit to the structure of our application. This allows us to reason about our component with high confidence. The unidirectional data flow as well as stateless nature makes it easy to understand, examine, and test our components. Also, the loose nature of inputs and outputs makes our component extremely portable. We can decide on a parent component, what data we'd like to run into our component, and how we'd like to handle changes.

Let's take a look at our `TaskList` component and how we change it to conform to our concept of "pure" components:

```
import {Component, ViewEncapsulation, Input, Output, EventEmitter,
ChangeDetectionStrategy} from '@angular/core';
import template from './task-list.html!text';
import {Task} from './task/task';
import {EnterTask} from './enter-task/enter-task';
import {Toggle} from '../ui/toggle/toggle';

@Component({
  selector: 'ngc-task-list',
  host: {
```

```
    class: 'task-list'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Task, EnterTask, Toggle],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class TaskList {
  @Input() tasks;
  // Event emitter for emitting an event once the task list has
  // been changed
  @Output() tasksUpdated = new EventEmitter();

  constructor() {
    this.taskFilterList = ['all', 'open', 'done'];
    this.selectedTaskFilter = 'all';
  }

  ngOnChanges(changes) {
    if (changes.tasks) {
      this.taskFilterChange(this.selectedTaskFilter);
    }
  }

  taskFilterChange(filter) {
    this.selectedTaskFilter = filter;
    this.filteredTasks = this.tasks ? this.tasks.filter((task) => {
        if (filter === 'all') {
          return true;
        } else if (filter === 'open') {
          return !task.done;
        } else {
          return task.done;
        }
      }) : [];
  }

  // Function to add a new task
  addTask(title) {
    const tasks = this.tasks.slice();
    tasks.push({ created: +new Date(), title, done: null });
    this.tasksUpdated.next(tasks);
  }
}
```

All the operations in our task list component are now immutable. We never directly modify our task's data that was passed in as input, but rather we create new task data arrays to perform mutable operations.

From what we've learned from the previous section, this effectively makes our component a "pure" component. This component itself is only relying on its input and makes our component very easy to reason about.

You've probably noticed that we've also configured the change-detection strategy of our

component. As we have a "pure" component now, we can configure our change-detection strategy accordingly to save some performance:

```
@Component({
  selector: 'ngc-task-list',
  …
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

As we're rendering a `Task` component for each data record in our task list, we should also check what we can change there in order to round this out.

Let's look at the changes in our `Task` component:

```
import {Component, Input, Output, EventEmitter, ViewEncapsulation, HostBinding,
ChangeDetectionStrategy} from '@angular/core';
import template from './task.html!text';
import {Checkbox} from '../../ui/checkbox/checkbox';

@Component({
  selector: 'ngc-task',
  host: {
    class: 'task'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Checkbox],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Task {
  @Input() task;
  // We are using an output to notify our parent about updates
  @Output() taskUpdated = new EventEmitter();

  @HostBinding('class.task--done')
  get done() {
    return this.task && this.task.done;
  }

  // We use this function to update the checked state of our task
  markDone(checked) {
    this.taskUpdated.next({
      title: this.task.title,
      done: checked ? +new Date() : null
    });
  }
}
```

We also use the `OnPush` strategy for our `Task` component, and we can do this because we also have a pure component here. This component only depends on its inputs. Both inputs expect native values (a `String` for title and a `Boolean` for done), which actually makes them immutable by nature. Changes on the task will be communicated using the `taskUpdated` output property.

Now, this is a good time to think about where to place our task list in the application. As we're writing a task management system that gives our users the ability to manage tasks within projects, we need to have a container that will encapsulate the concerns of projects. We create a new `Project` component, on the path `lib/project/project.js`, which will display project details and renders the `TaskList` component as a subcomponent:

```
import {Component, ViewEncapsulation, Input, Output, EventEmitter,
ChangeDetectionStrategy} from '@angular/core';
import template from './project.html!text';
import {TaskList} from '../task-list/task-list';

@Component({
  selector: 'ngc-project',
  host: {
    class: 'project'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [TaskList],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Project {
  @Input() title;
  @Input() description;
  @Input() tasks;
  @Output() projectUpdated = new EventEmitter();

  // This function should be called if the task list of the
  // project was updated
  updateTasks(tasks) {
    this.projectUpdated.next({
      title: this.title,
      description: this.description,
      tasks
    });
  }
}
```

Again, we make the state of this component dependent only on its immutable inputs, and we use the `OnPush` strategy in order to get the positive performance implications of using a pure component. It's also important to note that the `updateTasks` function acts as some sort of a delegate from our `TaskList` component. When we update a task inside the `TaskList` component, we catch the event in the project template and call the `updateTasks` function with the new updated task list. From here, we're just emitting the updated project data with the new task list further up in the component tree.

Let's also take a look at the template of our `Project` component quickly to understand the wiring behind this component:

```
<div class="project__l-header">
  <h2 class="project__title">{{title}}</h2>
  <p>{{description}}</p>
</div>
```

```
<ngc-task-list [tasks]="tasks"
               (tasksUpdated)="updateTasks($event)">
</ngc-task-list>
```

The binding logic in the template tells us how the whole data flow with our purified components work. While the `Project` component itself receives the list of tasks as input, it directly forwards this data to the `tasks` input of the `TaskList` component. If the `TaskList` component fires a `tasksUpdated` event, we're calling the `updateTasks` method on the `Project` component, which in fact just emits a `projectUpdated` event again.

Recap

The refactoring of our task list is now completed, and we applied our knowledge about immutable components and observable data structures to gain some performance wins in this structure. There won't be unnecessary dirty checking on our `Task` component any more because we switched to the `OnPush` change-detection strategy.

We have also reduced the complexity of the `TaskList` and `Task` components a lot, and it's now far easier to reason about these components and their state.

A further benefit of this refactoring is the great encapsulation level that we achieved using immutable inputs. Our `TaskList` component is not relying on any task container as a project. We can also pass it a list of tasks across all the projects, and it can still work as expected.

Composition using content projection

In this section, we will create a tabbed interface for our `Project` component that will help us further organize the structure of our application user interface. In order to create a `Tabs` component, we'll look at content projection and content child injection using observable query lists.

Input and output properties are great to establish encapsulation, and this is a main property of proper composition. However, sometimes the requirements are not to only pass data but to also pass content from the outside of a component into the component. In Shadow DOM, this is done using so-called slots. In Angular components, we can create content projection points using the `<ng-content>` element.

Let's look at a simple content projection example that helps us understand what this is good for:

```
@Component({
  selector: 'child',
  template: `
    <article>
      <header>
        <h1><ng-content select="[data-header]"></ng-content></h1>
      </header>
      <ng-content></ng-content>
    </article>
  `
})
export class Child {}

@Component({
  selector: 'app',
  template: `
    <child>
      <header data-header>Content projection is great</header>
      <p>Insert content in a controlled manner</p>
    </child>
  `,
  directives: [Child]
})
export class App {}
```

Looking at the `App` component in this example, we can see that we've put elements inside the actual `<child>` element. Usually, this content would be ignored and removed by Angular before it renders the `Child` component with its template.

However, when using content projection, the elements that are placed inside our component HTML element can be sucked into the `Child` component. This is what content projection is all about. Content projection is actually very similar to the concept of transclusion from Angular 1.

All that we need to do in order to enable content projection within a `Child` component is to place a `<ng-content>` element within its template. In this way, we specify at what locations in our component template we want to insert the content that is sucked in from the parent component.

Additionally, we can use the select attribute on the `<ng-content>` element to set a CSS-like selector. This selector will be used to only suck in specific elements, which match this selector. In this way, you can have multiple insertion points that cover different content requirements.

Elements from the component element can only be inserted once, and the content projection works by going through all the `<ng-content>` elements in sequential order by project matching elements. If you have multiple competing content projection points in your template that are interested in the same elements, the first one will actually win.

Creating a tabbed interface component

Let's introduce a new UI component in our `ui` folder in the project that will provide us with a tabbed interface that we can use for composition. We use what we learned about content projection in order to make this component reusable.

We'll actually create two components, one for `Tabs`, which itself holds individual `Tab` components.

First, let's create the component class within a new `tabs/tab` folder in a file called `tab.js`:

```
import {Component, Input, ViewEncapsulation, HostBinding} from '@angular/core';
import template from './tab.html!text';

@Component({
  selector: 'ngc-tab',
  host: {
    class: 'tabs__tab'
  },
  template,
  encapsulation: ViewEncapsulation.None
})
export class Tab {
  @Input() name;
  @HostBinding('class.tabs__tab--active') active = false;
}
```

The only state that we store in our `Tab` component is whether the tab is active or not. The name that is displayed on the tab will be available through an input property.

We use a class property binding to make a tab visible, based on the active flag we set a class; without this, our tabs are hidden.

Let's take a look at the `tab.html` template file of this component:

```
<ng-content></ng-content>
```

This is it already? Actually, yes it is! The `Tab` component is only responsible for the storage of its name and active state, as well as the insertion of the host element content in the content projection point. There's no additional templating that is needed.

Now, we'll move one level up and create the `Tabs` component that will be responsible for grouping all the `Tab` components. As we won't include `Tab` components directly when we want to create a tabbed interface but use the `Tabs` component instead, this needs to forward content that we put into the `Tabs` host element. Let's look at how we can achieve this.

In the `tabs` folder, we will create a `tabs.js` file that contains our `Tabs` component code, as follows:

```
import {Component, ViewEncapsulation, ContentChildren} from '@angular/core';
import template from './tabs.html!text';
// We rely on the Tab component
import {Tab} from './tab/tab';

@Component({
  selector: 'ngc-tabs',
  host: {
    class: 'tabs'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Tab]
})
export class Tabs {
  // This queries the content inside <ng-content> and stores a
  // query list that will be updated if the content changes
  @ContentChildren(Tab) tabs;

  // The ngAfterContentInit lifecycle hook will be called once the
  // content inside <ng-content> was initialized
  ngAfterContentInit() {
    this.activateTab(this.tabs.first);
  }

  activateTab(tab) {
    // To activate a tab we first convert the live list to an
    // array and deactivate all tabs before we set the new
    // tab active
    this.tabs.toArray().forEach((t) => t.active = false);
    tab.active = true;
  }
}
```

Let's observe what's happening here. We used a new `@ContentChildren` annotation, in order to query our inserted content for directives that match the type that we pass to the decorator. The `tabs` property will contain an object of the `QueryList` type, which is an observable list type that will be updated if the content projection changes. You need to remember that content projection is a dynamic process, as the content in the host element can actually change, for example, using the `NgFor` or `NgIf` directives.

We use the `AfterContentInit` lifecycle hook, which we've already briefly discussed in the *Custom UI elements* section of Chapter 2, *Ready, Set, Go!* This lifecycle hook is called after Angular has completed content projection on the component. Only then do we have the guarantee that our `QueryList` object will be initialized, and we can start working with child directives that were projected as content.

The `activateTab` function will set the `Tab` component's active flag, deactivating any previous active tab. As the observable `QueryList` object is not a native array, we first need to convert it using `toArray()` before we start working with it.

Let's now look at the template of the `Tabs` component that we created in a file called `tabs.html`

in the `tabs` directory:

```
<ul class="tabs__tab-list">
  <li *ngFor="let tab of tabs">
    <button class="tabs__tab-button"
            [class.tabs__tab-button--active]="tab.active"
            (click)="activateTab(tab)">{{tab.name}}</button>
  </li>
</ul>
<div class="tabs__l-container">
  <ng-content select="ngc-tab"></ng-content>
</div>
```

The structure of our `Tabs` component is as follows.

- First we render all the `tab` buttons in an unordered list.
- After the unordered list, we have a `tabs` container that will contain all our `Tab` components that are inserted using content projection and the `<ng-content>` element. Note that the selector that we use is actually the selector we use for our `Tab` component.
- Tabs that are not active will not be visible because we control this using CSS on our `Tab` component class attribute binding (refer to the `Tab` component code).

This is all that we need to create a flexible and well-encapsulated tabbed interface component. Now we can go ahead and use this component in our `Project` component to provide a segregation of our project detail information.

We will create three tabs for now, where the first one will embed our task list. We will address the content of the other two tabs in a later chapter.

Let's modify our `Project` component template in the `project.html` file as a first step.

Instead of including our `TaskList` component directly, we now use the `Tabs` and `Tab` components to nest the task list into our tabbed interface:

```
<ngc-tabs>
  <ngc-tab name="Tasks">
    <ngc-task-list [tasks]="tasks"
                   (tasksUpdated)="updateTasks($event)">
    </ngc-task-list>
  </ngc-tab>
  <ngc-tab name="Comments"></ngc-tab>
  <ngc-tab name="Activities"></ngc-tab>
</ngc-tabs>
```

You should have noticed by now that we are actually nesting two components within this template code using content projection, as follows:

- First, the `Tabs` component uses content projection to select all the `<ngc-tab>` elements. As these elements happen to be components too (our `Tab` component will attach to elements with this name), they will be recognized as such within the `Tabs` component once they are inserted.

- In the `<ngc-tab>` element, we then nest our `TaskList` component. If we go back to our `Task` component template, which will be attached to elements with the name `ngc-tab`, we will have a generic projection point that inserts any content that is present in the host element. Our task list will effectively be passed through the `Tabs` component into the `Tab` component.

Recap

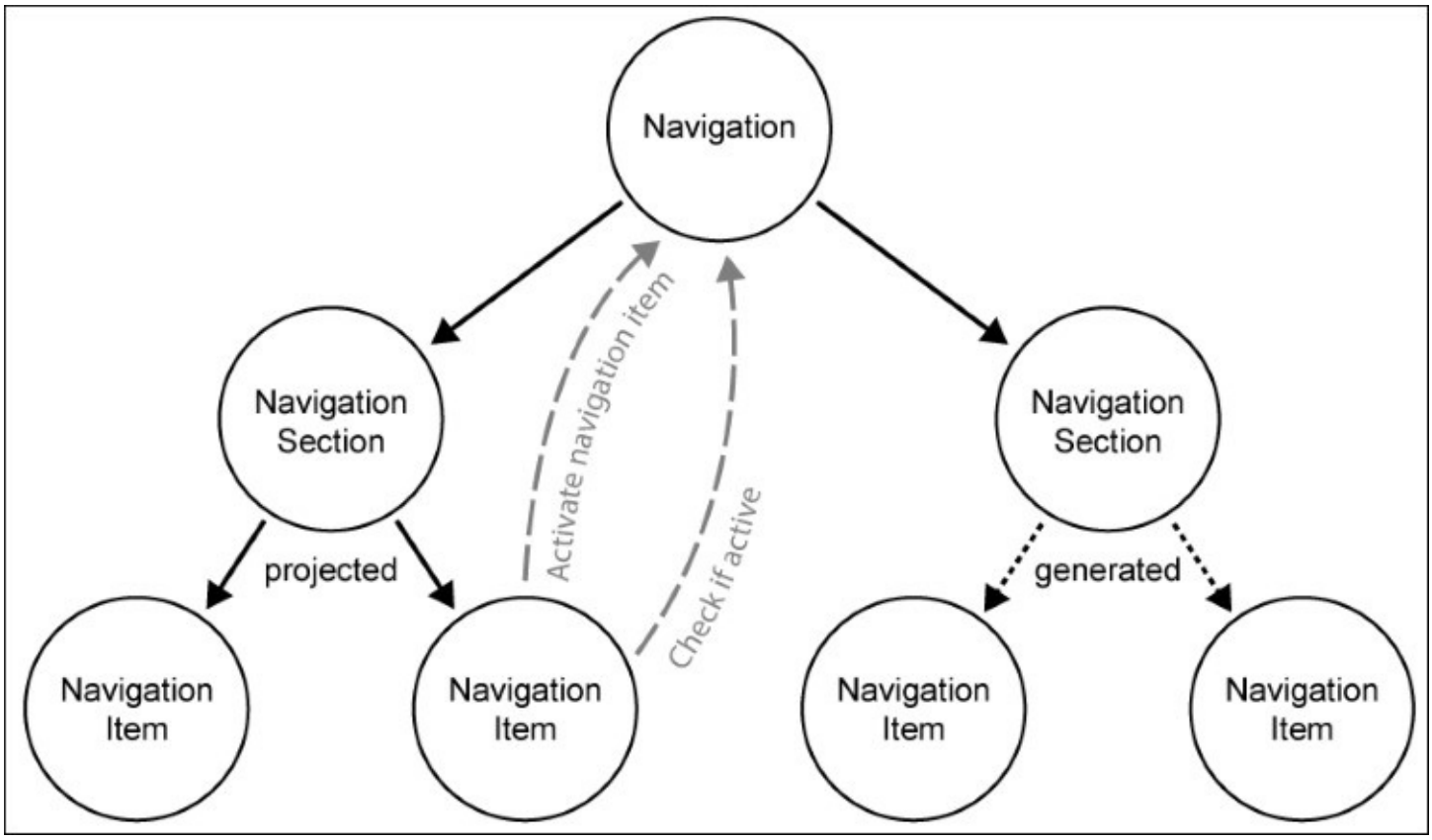
In this topic, we created a very handy tabbed interface component that we can use to segregate our user interface and provide a focused context for our users. We used content projection points using the `<ng-content>` elements. We also learned how to access inserted components using the `@ContentChildren` annotation and observable lists using the `QueryList` type.

Mixing projected with generated content

Our task management application supports the listing of multiple projects where a user can manage tasks. We need to provide a navigation that enables a user to browse through the existing projects. As projects come from our data store, the navigation will need to be generated dynamically. However, we also would like to have the possibility of specifying some navigation items within our navigation, as static content with pure templating.

In this section, we will create a simple navigation component, which will use content projection, so that we can add static navigation items. At the same time, navigation items can be generated from data and mixed with the static content-based navigation items.

Let's first take a look at an illustration of the architectural design and composition that we're going to use to implement our navigation:



An illustration of the navigation component tree and interactions

We'll use an intermediate component between the `Navigation` and `NavigationItem` components. The `NavigationSection` component is responsible for the division of multiple items into a section. The navigation sections also have a title that will be displayed on top of the item list.

The illustration shows two `NavigationSection` components, where the left one uses pure content projection to create items, as we have learned in the previous section. The right `NavigationSection` component generates items using an input data structure, which is a list of navigation item models.

As we have intermediate components between the `Navigation` and `NavigationItems` components (we can only have one selected navigation), we also establish a direct communication path between them. We will achieve this using ancestor component injection.

Note

The architectural approach for this navigation is just one of many possible approaches. We choose this approach in order to show you how we can easily mix content projection and generated content. In this example, we don't use the Angular router to provide navigation state and route mapping. This will be part of a later chapter.

Let's start bottom up with the `NavigationItem` component and create a new `navigation-item.js` file in a newly-created `navigation/navigation-section/navigation-item` path:

```
// We rely on the navigation component to know if we are active
import {Navigation} from '../../navigation';

@Component({
  selector: 'ngc-navigation-item'
})
export class NavigationItem {
  @Input() title;
  @Input() link;

  constructor(@Inject(Navigation) navigation) {
    this.navigation = navigation;
  }

  // Here, we are delegating to the navigation component to see if
  // we are active or not
  isActive() {
    return this.navigation.isItemActive(this);
  }

  // If this link is activated we need to tell the navigation component
  onActivate() {
    this.navigation.activateLink(this.link);
  }
}
```

From the `NavigationItem` component code, we can see that we're directly communicating with the `Navigation` ancestor component. We can simply inject the `NavigationComponent`, as this is a child of the component. As the `Navigation` items will never exist without a `Navigation` component, we should be fine with this direct dependency.

Let's move on to the `NavigationSection` component that is the intermediate component

between the `Navigation` component and the items and is responsible for the grouping of items together.

We will create a file called `navigation-section.js` in the `navigation/navigation-section` path:

```
@Component({
  selector: 'ngc-navigation-section',
  directives: [NavigationItem]
})
export class NavigationSection {
  @Input() title;
  @Input() items;
}
```

Hold on! That's all that this needs? Didn't we say that we want our `NavigationSection` component to also be responsible for not only providing a way to insert content, but also accepting data in order to generate items? Well, this is true. However, this is actually pure templating logic, and it can be done solely within the template file of the component. All that we need is an optional input with item data that we will use to generate the `NavigationItem` components.

Let's create the view template for this component in a file named `navigation-section.html`:

```
<h2 class="navigation-section__title">{{title}}</h2>
<ul class="navigation-section__list">
  <ng-content select="ngc-navigation-item"></ng-content>
  <ngc-navigation-item *ngFor="let item of items"
                       [title]="item.title"
                       [link]="item.link"></ngc-navigation-item>
</ul>
```

Well, this wasn't rocket science, was it? However, this shows the great flexibility that we have in Angular component templates:

- Firstly, we create a content projection point that selects all the elements from the host element that match the name `ngc-navigation-item`. This means that the `NavigationItem` components can be placed outside the component in a very static fashion to create, for example, static links. As the model properties of navigation items are directly exposed as bindable attributes on the `NavigationItem` element, we can also place them statically into a pure HTML template with regular DOM attributes.
- Secondly, we can use the `NgFor` directive to generate the `NavigationItem` components inside the component. Here, we just iterate over the list of navigation item models that acts as an optional input to our component. We use bindings in the items model so that we can even propagate change into our navigation item components.

As a final step, we create the `Navigation` component itself that uses content projection points so that we can manage the `NavigationSection` component from outside. We create a file called `navigation.js` to write the code of the `Navigation` component:

```
import {NavigationSection} from './navigation-section/navigation-section';

@Component({
  selector: 'ngc-navigation',
  directives: [NavigationSection]
})
export class Navigation {
  @Input() activeLink;

  // Checks if a given navigation item is currently active by its
  // link. This function will be called by navigation item child
  // components.
  isItemActive(item) {
    return item.link === this.activeLink;
  }

  // If a link wants to be activated within the navigation, this
  // function needs to be called. This way child navigation item
  // components can activate themselves.
  activateLink(link) {
    this.activeLink = link;
    this.activeLinkChange.next(this.activeLink);
  }
}
```

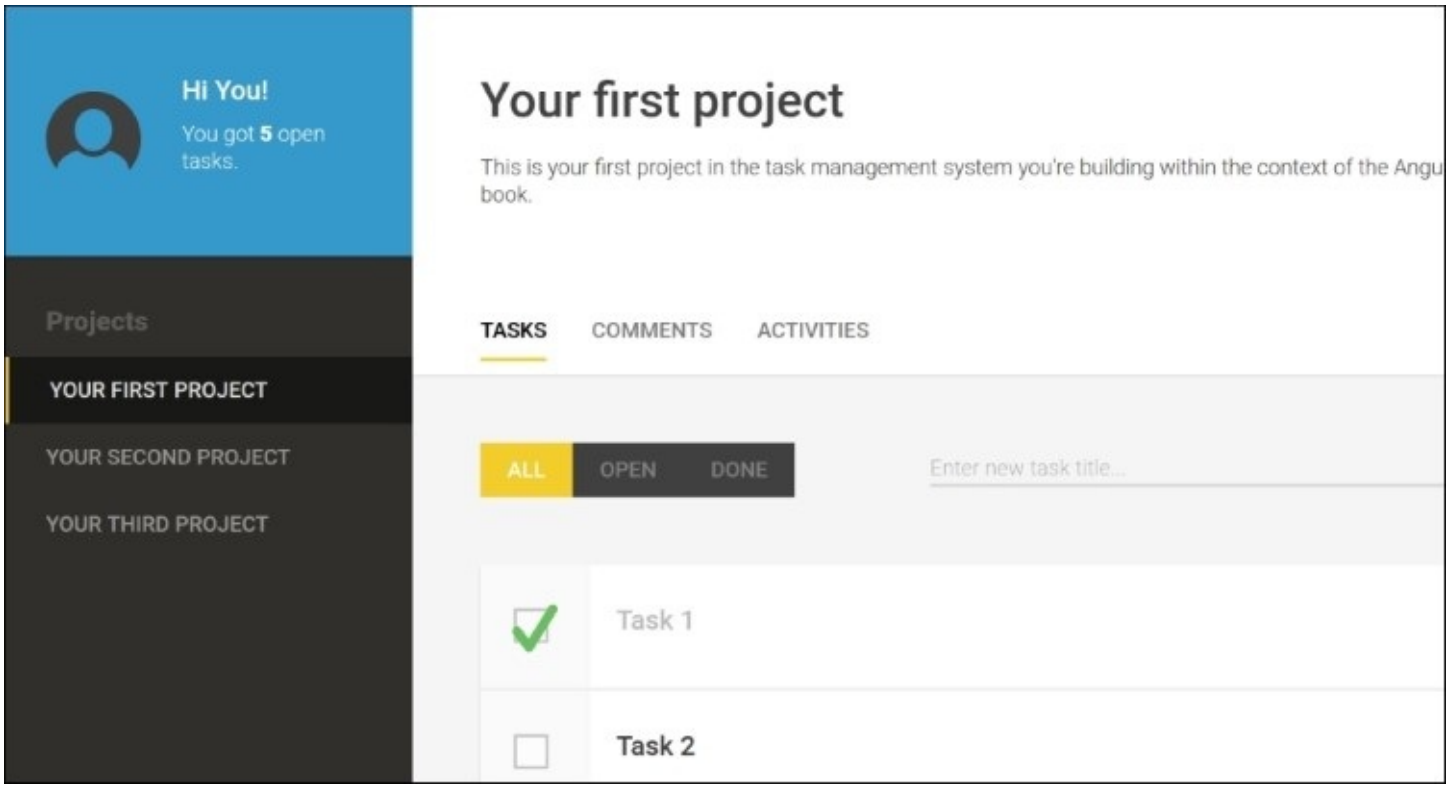
In the `Navigation` component, we store the state of which navigation item is activated. This is also provided as input to the component so that we can set the activated link with an input binding from outside. The `isItemActive` and `activateLink` functions are there to monitor and change the state of the active item within the navigation. These functions are directly used within the `NavigationItem` components, which inject the navigation using ancestor component injection.

Now, the only bit that is missing is to include our navigation in the main application. For this, we will edit the `app.html` template of the component:

```
<div class="app">
  <div class="app__l-side">
    <ngc-navigation
            [activeLink]="getSelectedProjectLink()"
            (activeLinkChange)="selectProjectByLink($event)">
      <ngc-navigation-section
                  title="Projects"
                  [items]="getProjectNavigationItems()">
      </ngc-navigation-section>
    </ngc-na
vigation>
  </div>
  <div class="app__l-main">
    …
  </div>
</div>
```

Here, we only use the generative approach to write a `NavigationSection` component where

we actually pass a list of navigation item models into the navigation component. This list is generated by the `getProjectNavigationItems` function on our main application component using the available projects from our observable data structure:



A screenshot of the newly-created project navigation

Summary

In this chapter, we learned about how we can profit from concepts, such as reactive programming, observable data structures, and immutable objects, in order to make our application perform better, and most importantly, simple and easy to reason about.

We touched on the different change-detection strategies and learned how to use the `OnPush` strategy to gain better performance in combination with immutable data.

We built a tabbed user interface component that we can reuse wherever we need it, and we learned about the concept of content projection. We also created a simple navigation component tree that uses a mix of content projection and generation. The navigation items also directly communicate with their ancestor `Navigation` component, in order to manage their state using ancestor component injection.

As we switched to a reactive approach to manage data within our application, I want you to perform a little experiment. If you've downloaded the final chapter's code, go ahead and open two browser windows that point to the task management application. You will be amazed that we already have a working real-time synchronization in place that allows us to work in both browser windows and have both of them updated at the same time. This has all been made possible because of the reactive and functional way that we work with data in our components.

Chapter 4. No Comments, Please!

During the course of this chapter, we will create reusable components to enable commenting not only on projects, but also on any other entity within our application. We'll build our commenting system in a way that it will allow us to place it anywhere we'd like for our users to put comments. In order to provide our users with a feature to edit existing comments and also a seamless authoring experience, we'll create an editor UI component that could be used to make arbitrary content within our application editable.

Discussing security and proper user management in this chapter is still out of scope, but we're going to create a dummy user service that will help us simulate a logged-in user. This service will be used by the commenting system, and we'll refactor our existing component to make use of it too.

We'll cover the following topics in this chapter:

- Using `contenteditable` to create an in-place editor
- Using `@HostBinding` and `@HostListener` to bind component members to host element properties and events
- Communicating directly with view children using the `@ViewChild` annotation
- Performing DOM operations by injecting and using `ElementRef`
- Creating a dummy user service and using the `@Injectable` annotation to serve it as a dependency injection provider
- Applying custom actions on component input changes, using the `OnChanges` life cycle hook
- Creating a simple pipe to format relative time intervals using the `Moment.js` library

One editor to rule them all

Since we will be processing a lot of user input within our application, it's crucial to provide a nice authoring experience to our users. Within the commenting system we're about to create in this chapter, we need a way through which users could edit existing comments, as well as add new comments. We could use regular text area input and work with dialog boxes to edit comments, but this seems too old-fashioned for a modern user interface, which we're going to build, and does not really provide a great user experience. What we're looking for is a way to edit stuff in place. The commenting system will not only benefit from such an in-place editor, but it will also help us create the editor component in such a way that we can use it for any content within our application that we'd like to make editable.

In order to build our in-place editor, we're going to use the `contenteditable` API that will enable a user to modify the content within the HTML elements directly in the site document.

The following example illustrates how we can use the `contenteditable` attribute to make HTML elements editable:

```
<h1 contenteditable>I'm an editable title</h1>
<p>I can't be edited</p>
```

Run the preceding example on a blank HTML page and click on the `h1` text. You will see that the element has become editable and you can type to modify its content.

Getting notified about changes within editable elements is fairly easy. There's an input event emitted on every DOM element that is editable, and this will allow us to react to a change easily:

```
const h1 = document.querySelector('h1');
h1.addEventListener('input',(event)=>console.log(h1.textContent);
```

With this example, we have already created a naive implementation of an in-place editor where we're able to monitor changes applied by the user. Within this topic, we'll use this standard technology to build a reusable component that we can use wherever we want to make things editable.

Creating an editor component

First, let's create a new folder named `editor` within our `ui` folder. In this folder, we're going to create a new component file named `editor.js`:

```
import {Component, ViewChild, Input, Output, ViewEncapsulation, EventEmitter,
HostBinding, HostListener} from '@angular/core';
import template from './editor.html!text';

@Component({
  selector: 'ngc-editor',
  host: {
    class: 'editor'
  },
  template,
  encapsulation: ViewEncapsulation.None
})
export class Editor {
  // Using view child reference with local view variable name
  @ViewChild('editableContentElement') editableContentElement;
  // Content that will be edited and displayed
  @Input() content;
  // Creating a host element class attribute binding from the
  // editMode property
  @Input() @HostBinding('class.editor--edit-mode') editMode;
  @Input() showControls;
  @Output() editSaved = new EventEmitter();
  @Output() editableInput = new EventEmitter();

  // We need to make sure to reflect to our editable element if
  // content gets updated from outside
  ngOnChanges() {
    if (this.editableContentElement && this.content) {
      this.setEditableContent(this.content);
    }
  }

  ngAfterViewInit() {
    this.setEditableContent(this.content);
  }

  // This returns the content of our content editable
  getEditableContent() {
    return this.editableContentElement.nativeElement.textContent;
  }

  // This sets the content of our content editable
  setEditableContent(content) {
    this.editableContentElement.nativeElement.textContent =
      content;
  }

  // This annotation will create a click event listener on the
  // host element that will invoke the underlying method
  @HostListener('click')
```

```
  focusEditableContent() {
    if (this.editMode) {
      this.editableContentElement.nativeElement.focus();
    }
  }

  // Method that will be invoked if our editable element is
  // changed
  onInput() {
    // Emit a editableInput event with the edited content
    this.editableInput.next(this.getEditableContent());
  }

  // On save we reflect the content of the editable element into
  // the content field and emit an event
  save() {
    this.editSaved.next(this.getEditableContent());
    this.setEditableContent(this.content);
    // Setting editMode to false to switch the editor back to
    // viewing mode
    this.editMode = false;
  }

  // Canceling the edit will not reflect the edited content and
  // switch back to viewing mode
  cancel() {
    this.setEditableContent(this.content);
    this.editableInput.next(this.getEditableContent());
    this.editMode = false;
  }

  // The edit method will initialize the editable element and set
  // the component into edit mode
  edit() {
    this.editMode = true;
  }
}
```

Okay, that's quite a lot of new code. Let's dissect the different parts of the `Editor` component and go through each part step by step.

Within our `Editor` component, we'll need to interact with the native DOM element, which is editable. The easiest and also the safest method to do this is to use the `@ViewChild` decorator in order to retrieve an element with a local view reference:

```
@ViewChild('editableContentElement') editableContentElement;
```

In the previous chapter, we learned about the `@ContentChildren` annotation, which helps us obtain a list of all the child components within content projection points. If we would like to do the same with regular view children, we need to use the equivalent `@ViewChildren` annotation. While `@ContentChildren` searches for components within content projection points, `@ViewChildren` hunts for the regular sub-tree of a component.

If we want to search the component sub-tree for one single component, we can use the `@ViewChild` annotation (please note that `@ViewChild` and `@ViewChidren` are different).

Query annotation	Description
`@ViewChildren`(selector)	Will query the current component's view for either directives or components and return an object of the type `QueryList`. If the view is dynamically updated, this list will be updated as well.
`@ViewChild`(selector)	Will query for only the first matching component or directive and return an instance of it.

Note

A selector can be either a directive or component type, or a string that contains the name of a local view variable. If a local view variable name is provided, Angular will search for the element containing the view variable reference.

If you need to communicate with view child components directly, using `@ViewChild` and `@ViewChildren` annotations should be your preferred way.

Tip

Sometimes you need to run the initialization code on view children after your component is initialized. In such cases, you can use the `AfterViewInit` life cycle hook. While the view child properties of your component class will still be undefined within the constructor of your component, they will be populated and initialized after the `AfterViewInit` life cycle callback.

The `@ViewChild` and `@ViewChildren` decorators are great tools to interact with in your view directly. It doesn't really matter whether you'd like to interact with a DOM element or a component instance. Both use cases are nicely covered using this declarative API.

Let's move back to our `Editor` component code. The next thing we're going to look into are the component's input functions:

```
@Input() content;
@Input() @HostBinding('class.editor--edit-mode') editMode;
@Input() showControls;
```

The `content` input property is the main interface for interacting with the component from outside. Using property bindings, we can have any preexisting text content set up in the editor component.

The `editMode` property is a Boolean value that controls whether the editor is in edit or display

mode. Our editor component will depend on this flag to know whether content should be edited or not. This allows us to switch from read-only mode to edit mode and back interactively.

Though an input property, this flag can be controlled from outside the component. At the same time, it can also be used to create property binding of a host element. Specifically, we can use the flag to create a class attribute binding to add or remove the modifier class, `editor--edit-mode`. This class is used to control some differences in the visual appearance of the editor while in edit mode.

The last of the three input properties in our editor component, `showControls`, controls whether the editor should show the control functions. There are three controls that will be shown when this property evaluates to a true value:

- **Edit button**: This will be shown when the component is in display mode, and it will switch the component to edit mode using the `editMode` flag.
- **Save button**: This will be shown only if the component is in edit mode. This control will save the changes applied within the current edit mode and switch the component back to display mode.
- **Cancel button**: This is the same as the save button, and this control is shown only when the component is in edit mode. If activated, the component will switch back to display mode, reverting any changes that you may have made.

Besides our input properties, we also need some output properties to notify the outer world about the changes within our editor. The following piece of code helps us do this:

```
@Output() editSaved = new EventEmitter();
@Output() editableInput = new EventEmitter();
```

The `editSaved` event will be emitted once the edited content is saved using the save button control. Also, it'll be better if an event is emitted upon every input change within our editable content element. For this, we used the `editableInput` output property.

Our editor component works in a simple way. If the component is in edit mode, it shows an element that can be edited. However, once the editor switches back to display mode, we see a different element that cannot be edited. The visibility is controlled with the modifier class set by the host element property binding to the `editMode` flag.

Angular has no control over the content within our editable element. We control this content manually by using native DOM operations. Let's look at how we did this. First of all, we needed to use delegates to access the element, since we're most likely going to change how we will read and write to and from the editable element. We used the following to do this:

```
getEditableContent() {
  return this.editableContentElement.nativeElement.textContent;
   }

setEditableContent(content) {
```

```
  this.editableContentElement.nativeElement.textContent =
    content;
}
```

Tip

Note that we used the `nativeElement` property on our `editableContentElement` field, previously set by the `@ViewChild` decorator.

Angular does not directly provide us with a DOM element reference but a wrapper object of the type `ElementRef`. It's basically a wrapper around the native DOM element, holding additional information that is relevant to Angular.

Using the `nativeElement` accessor, we can obtain a reference to the underlying DOM element.

Note

The `ElementRef` wrapper plays an important part in Angular's platform-agnostic architecture. It allows you to run Angular in different environments (for example, native mobile, web workers, or others). It's part of an abstraction layer between the components and their views.

We also needed a way to set the content of the editable element based on the input that we would receive from the content input property. We could use the life cycle hook `OnInit`, which will be called only after the input properties are checked upon component initialization. However, this life cycle hook fires only once after the initialization, and we needed a way that would have helped us react to subsequent input changes of the `content` property. Have a look at the following code snippet:

```
ngOnChanges() {
  if (this.editableContentElement && this.content) {
    this.setEditableContent(this.content);
  }
}
```

The `OnChanges` life cycle hook is exactly what we needed here. With this, once a change in the content input property is detected (this also includes the first change after the initialization), we can reflect the changed content onto our editable element.

Now we have already implemented the reflection of the component content input property onto the editable field. But what about the opposite direction? We need to find a way to reflect the changes in our editable element onto our component `content` property. That's also closely related to the actions performed on the component using the available controls within edit mode, which are as follows:

- **In the save operation**: Here, we reflect the edited content from the editable element back to the component's `content` property.
- **In the cancel operation**: Here, we ignore what has been edited by the user within the editable element and set its content back to the value in the component's `content`

property:

Let's look at the code for those two operations:

```
save() {
  this.editSaved.next(this.getEditableContent());
  this.setEditableContent(this.content);
  this.editMode = false;
}

cancel() {
  this.setEditableContent(this.content);
  this.editableInput.next(this.getEditableContent());
  this.editMode = false;
}
```

In addition to the highlighted code, which shows the reflection between the component's `content` property and the editable element, we emitted certain events that would help us notify the outside world about the changes. In both the operations, we set the `editMode` flag to `false` after completion. This ensures that our editor will switch to display mode after any one of the operations is completed.

The `edit` method will be called from the edit control button when the component is in display mode. The only thing it does is that it switches the component back to edit mode:

```
edit() {
  this.editMode = true;
}
```

Whatever we've discussed thus far in relation to the code is good enough for us to set up a fully functional component. However, the last part of the code, which we haven't discussed yet, relates to ensuring better accessibility of our editor. Since our editor component is a bit larger than the editable element, we also want to make sure that a click anywhere inside the editor component will cause the editable element to be focused. The following code makes this happen:

```
@HostListener('click')
focusEditableContent() {
  if (this.editMode) {
    this.editableContentElement.nativeElement.focus();
  }
}
```

Using the `@HostListener` decorator, we registered an event binding on our component element that called the `focusEditableContent` method. Inside this method, we used the reference to the editable DOM element and triggered a focus.

Let's look at the template of our component that is located within the `editor.html` file in order to see how we could interact with the logic within our component:

```
<div (input)="onInput($event)"
```

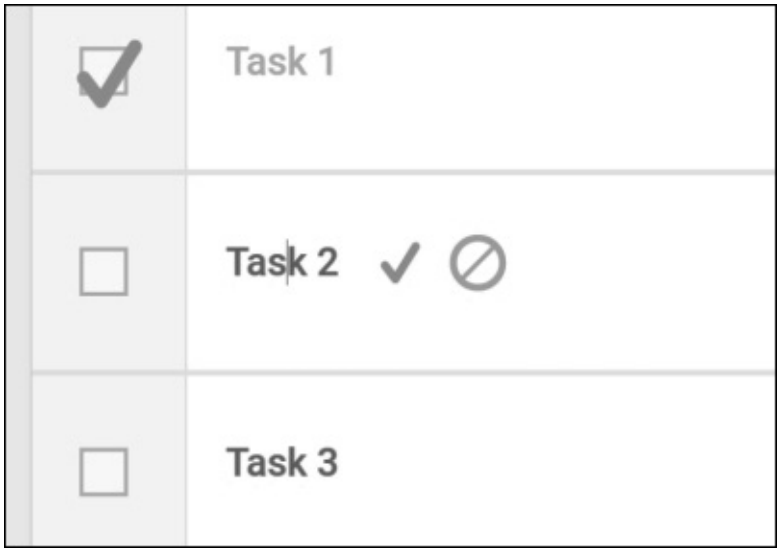
```
      class="editor__editable-content"
      contenteditable="true"
      #editableContentElement></div>
<div class="editor__output">{{content}}</div>
<div *ngIf="showControls && !editMode" class="editor__controls">
  <button (click)="edit()" class="editor__icon-edit"></button>
</div>
<div *ngIf="showControls && editMode" class="editor__controls">
  <button (click)="save()" class="editor__icon-save"></button>
  <button (click)="cancel()" class="editor__icon-cancel"></button>
</div>
```

The logic within the editor component template is quite straightforward. If you've been following the component code, you'll now be able to identify the different elements that compose this component's view.

The first element with the `editor__editable-content` class is our editable element that has the `contenteditable` attribute. The user will be able to type into this element when the editor is in edit mode. It's important to note that we've annotated it with a local view variable reference, `#editableContentElement`, which we're using in our view child queries.

The second element with the `editor__output` class is only to display the editor content and is only visible when the editor is in display mode. The visibility of both the elements is controlled using CSS, based on the `editor--edit-mode` modifier class, which, if you recall from the component class code, is set through host property binding based on the `editMode` property.

The three control buttons are shown using the `NgIf` directive conditionally. The `showControls` input property needs to be `true`, and depending on the `editMode` flag, the screen will either show the edit button or the save and the cancel button:



Screenshot of our editor component in action

Recap

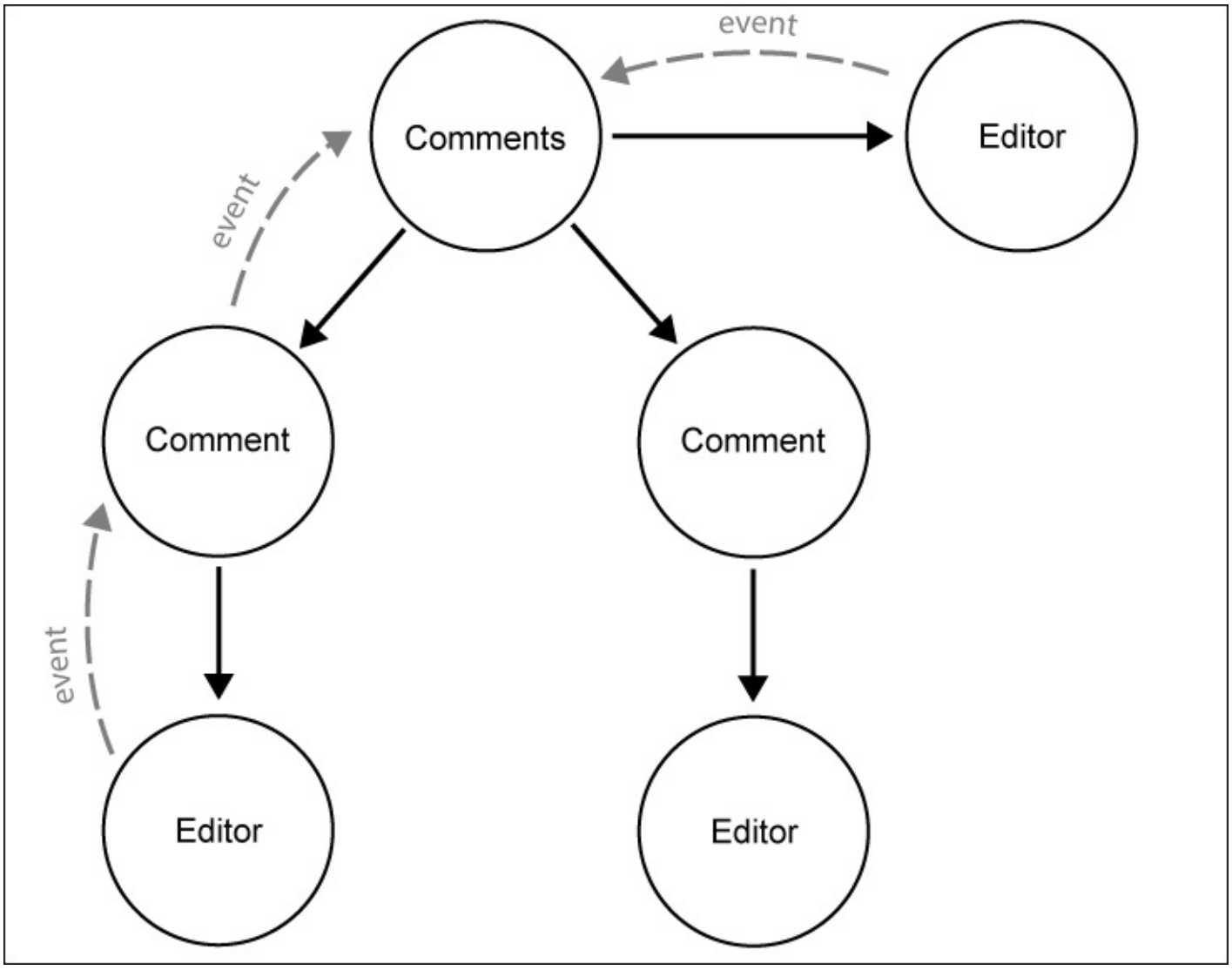
Within this building block, we have created an in-place editor widget, which we can use to grab user input for any content within our application. It allows us to provide the user with contextual editing capabilities, which will result in a great user experience.

We have also learned about the following topics:

1. Using `contenteditable` HTML5 attribute to enable in-place editing.
2. Using `@ViewChild` and `@ViewChildren` to query view child elements.
3. Using the `ElementRef` dependency to perform native DOM operations.
4. Implementing the logic, using the `OnChange` life cycle hook, to reflect data between Angular and the content that is not in immediate control of Angular.

Building a commenting system

In the previous topic, we created an editor component that will support users in editing content within our application. Here, we're going to create a commenting system that will enable users to write comments in various areas of our application. The commenting system will use our editor component to make comments editable, and thereby help users create new comments:



An illustration of the component sub-tree of a commenting system

The preceding diagram illustrates the architecture of the component tree within the commenting system that we are about to create.

The `Comments` component will be responsible for listing all the existing comments, as well as creating new comments.

Each comment itself is encapsulated into a `Comment` component. `Comment` components themselves use an editor that enables users to edit comments once they are created.

The `Editor` component, which we built in the previous topic, is used by the `Comment` component directly, to provide an input control for adding new comments. This allows us to reuse the functionality of our editor component to capture user input.

The `Editor` component emits an `editSaved` event once editable content is saved using the control buttons of the editor. In the `Comment` component, we will capture these events and propagate a new event upward to our `Comments` component. There, we will do the necessary updates but then again emit a new event to notify our parent about the change. In a composition of components, each component will react on change and delegate to the parent component if necessary.

Building the comment component

Let's start building our commenting system by fleshing out the `Comment` component first. In addition to the comment itself, we'd like to display the user's profile who commented, and of course, the time of the comment.

To display the time, we will make use of relative time formatting, as this will give our users a better feel of time. Relative time formatting displays timestamps in the format "5 minutes ago" or "1 month ago", in contrast to absolute timestamps, such as "25.12.2015 18:00". Using the `Moment.js` library, we'll create a pipe that we can use within component templates to convert timestamps and dates into relative time intervals.

Let's create a new pipe within a new folder named `pipes`. The pipe needs to be created within a file named `from-now.js`, which is created under the `pipes` folder:

```
import {Pipe} from '@angular/core';
// We use the Moment.js library to convert dates to relative times
import Moment from 'moment';

@Pipe({
  // Specifying the name to be used within templates
  name: 'fromNow'
})
// Our pipe will transform dates and timestamps to relative times
// using Moment.js
export class FromNowPipe {
  // The transform method will be called when the pipe is used
  // within a template
  transform(value) {
    if (value && (value instanceof Date ||
        typeof value === 'number')) {
      return new Moment(value).fromNow();
    }
  }
}
```

This pipe can now be used within the templates of components to format timestamps and dates into relative time intervals.

Let's use this pipe and the `Editor` component we created in the previous topic to create our `Comment` component. Within a file named `comment.html`, which is located within a new `comment` folder in the `comments` folder, we'll create the template for our `Comment` component:

```
<div class="comment__l-meta">
  <div class="comment__user-picture">
    <img [attr.src]="user.pictureDataUri" src="">
  </div>
  <div class="comment__user-name">{{user.name}}</div>
  <div class="comment__time">
    {{time | fromNow}}
  </div>
```

```
</div>
<div class="comment__l-main">
  <div class="comment__message">
    <ngc-editor [content]="content"
                [showControls]="true"
                (editSaved)="onContentSaved($event)">
    </ngc-editor>
  </div>
</div>
```

From the user object, we will get the user's profile image as well as the username. To display the time of the comment in a relative format, we'll use the `fromNow` pipe that we created earlier.

Finally, we will make use of the in-place editor component to display the content of the comment and make it editable at the same time. We will bind the comment content property to the content input property of the editor. At the same time, we will listen for the `editSaved` event of the editor and call the `onContentSaved` method on our comment component class. If you look at our component code again, you'll notice that we are re-emitting the event within the method so that the outside world is also notified about the change in the comment.

Let's take a look at the component class that we will create in a file named `comment.js`:

```
import {Component, Input, Output, ViewEncapsulation, EventEmitter} from
'@angular/core';
import {Editor} from '../../ui/editor/editor';
import template from './comment.html!text';
// We use our fromNow pipe that converts timestamps to relative
// times
import {FromNowPipe} from '../../pipes/from-now';

@Component({
  selector: 'ngc-comment',
  host: {
    class: 'comment'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Editor],
  pipes: [FromNowPipe]
})
export class Comment {
  // The time of the comment as a timestamp
  @Input() time;
  // The user object of the user who created the comment
  @Input() user;
  // The comment content
  @Input() content;
  // If a comment was edited this event will be emitted
  @Output() commentEdited = new EventEmitter();

  onContentSaved(content) {
    this.commentEdited.next(content);
  }
}
```

The component code is pretty straightforward. The only noticeable difference to other components we've created so far is the `pipes` property within the component's annotation. Here, we specify that we'd like to use the `FromNowPipe` class that we've just created. Pipes always need to be declared within the component; otherwise, they can't be used within the component's template.

As input, we expect a user object that is passed along with the `user` input property. The `content` input property should be filled with the actual comment as a string, while the `time` input property should be set to a timestamp that reflects the actual time of the comment.

We also have an output property called `commentEdited`, which we will use to notify the changes on the comment. The `onEditSaved` method will be called by the event binding on our `Editor` component, which will then emit an event using the `commentEdited` output property.

Building the comments component

We now have all the components ready in order to finish building our commenting system. The last missing piece of the puzzle is the `Comments` component, which will list all the comments and provide an editor to create new comments.

First, let's take a look at the template of our `Comments` component that we will create in a file named `comments.html` within a folder named `comments`:

```
<div class="comments__title">Add new comment</div>
<div class="comments__add-comment-section">
  <div class="comments__add-comment-box">
    <ngc-editor [editMode]="true"
                [showControls]= "false"></ngc-editor>
  </div>
  <button (click)="addNewComment()"
          class="button" >Add comment</button>
</div>

<div *ngIf="comments?.length > 0">
  <div class="comments__title">All comments</div>
  <ul class="comments__list">
    <li *ngFor="let comment of comments">
      <ngc-comment [content]="comment.content"
              [time]="comment.time"
              [user]="comment.user"
              (commentEdited)="onCommentEdited(comment, $event)">
      </ngc-comment>
    </li>
  </ul>
</div>
```

You can see the direct usage of an `Editor` component within the component's template. We are using this in-place editor to provide an input component to create new comments. We could also use a text area here, but we've decided to reuse our `Editor` component. We will set the `editMode` property to true so it will be initialized in edit mode. We will also set the `showControls` input to `false` because we don't want the editor to become autonomous. We will only use its in-place editing capabilities, but control it from our `Comments` component.

To add a new comment, we will use a button that has a click event binding, which calls the `addNewComment` method on our component class.

Below the section where users can add new comments, we will create another section that will list all the existing comments. If no comments exist, we simply don't render the section. With the help of the `NgFor` directive, we could display all the existing comments and create a `Comment` component for each repetition. We will bind all the comment data properties to our `Comment` component and also add an event binding to handle updated comments.

Let's create the component class within a new file named `comments.js` in the `comments` folder:

```
import {Component, Inject, Input, Output, ViewEncapsulation, ViewChild,
EventEmitter} from '@angular/core';
import template from './comments.html!text';
import {Editor} from '../ui/editor/editor';
import {Comment} from './comment/comment';
import {UserService} from '../user/user-service/user-service';

@Component({
  selector: 'ngc-comments',
  host: {
    class: 'comments'
  },
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Comment, Editor]
})
export class Comments {
  // A list of comment objects
  @Input() comments;
  // Event when the list of comments have been updated
  @Output() commentsUpdated = new EventEmitter();
  // We are using an editor for adding new comments and control it
  // directly using a reference
  @ViewChild(Editor) newCommentEditor;

  // We're using the user service to obtain the currently logged
  // in user
  constructor(@Inject(UserService) userService) {
    this.userService = userService;
  }

  // We use input change tracking to prevent dealing with
  // undefined comment list
  ngOnChanges(changes) {
    if (changes.comments &&
        changes.comments.currentValue === undefined) {
      this.comments = [];
    }
  }

  // Adding a new comment from the newCommentContent field that is
  // bound to the editor content
  addNewComment() {
    const comments = this.comments.slice();
    comments.splice(0, 0, {
      user: this.userService.currentUser,
      time: +new Date(),
      content: this.newCommentEditor.getEditableContent()
    });
    // Emit event so the updated comment list can be persisted
    // outside the component
    this.commentsUpdated.next(comments);
    // We reset the content of the editor
    this.newCommentEditor.setEditableContent('');
  }
```

```
  // This method deals with edited comments
  onCommentEdited(comment, content) {
    const comments = this.comments.slice();
    // If the comment was edited with e zero length content, we
    // will delete the comment from the list
    if (content.length === 0) {
      comments.splice(comments.indexOf(comment), 1);
    } else {
      // Otherwise we're replacing the existing comment
      comments.splice(comments.indexOf(comment), 1, {
        user: comment.user,
        time: comment.time,
        content
      });
    }
    // Emit event so the updated comment list can be persisted
    // outside the component
    this.commentsUpdated.next(comments);
  }
}
```

Let's go through individual code parts again and discuss what each of them does. First, we declared an input property named `comments` in our component class:

```
@Input() comments;
```

The `comments` input property is a list of comment objects that contains all of the data associated with the comments. This includes the user who authored the comment and the timestamp, as well as the content of the comment.

We also need to be able to emit an event once a comment is added or an existing comment is modified. For this purpose, we used an output property named `commentsUpdates`:

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

Once a new comment is added or an existing one is modified, we will emit an event from this output property with the updated list of comments.

The `Editor` component we're going to use to add new comments will not have its own control buttons. We will use the `showControls` input property to disable them. Instead, we will control the editor from our `Comments` component directly. Therefore, we need a way to communicate with the `Editor` component within our component class.

We used the `@ViewChild` decorator for this purpose again. However, this time, we did not reference a DOM element, which contains a local view variable reference. We directly passed our component type class to the decorator. Angular will search for any `Editor` components within the comments view and provide us with a reference to the instance of the editor. This is shown in the following line of code:

```
@ViewChild(Editor) newCommentEditor;
```

Since the `Comments` component only hosts one editor directly within the component template, we can use the `@ViewChild` annotation to obtain a reference to it. Using this reference, we can directly interact with the child component. This will allow us to control the editor directly from our `Comments` component.

Let's move on to the next part of the code, which is the `Comments` component constructor. The only thing we've done here is inject a user service that will provide us with a way to obtain information of the currently logged-in user. As of now, this functionality is only mocked, and we will receive information of a dummy user. We need this information in the `Comments` component, since we need to know which user has actually entered a new comment:

```
constructor(@Inject(UserService) userService) {
  this.userService = userService;
}
```

In the next part of the code, we controlled how we should react to the changes of the `comments` input property. Actually, we would never want the list of comments to remain undefined. It should be an empty list in case there are no comments, but the input property comments should never be undefined. We controlled this by using the `OnChange` life cycle hook and overriding our `comments` property if it was set to `undefined` from outside:

```
ngOnChanges(changes) {
  if (changes.comments &&
          changes.comments.currentValue === undefined) {
    this.comments = [];
  }
}
```

This small change makes the internal handling of our comment data much cleaner. We don't need additional checks when working for array transformation functions, and we can always treat the `comments` property as an array.

Since the `Comments` component is also responsible for handling the logic that deals with the process of adding new comments, we needed a method that could implement this requirement. In relation to this, we used some immutable practices we learned about in the previous chapter:

```
addNewComment() {
  const comments = this.comments.slice();
  comments.splice(0, 0, {
    user: this.userService.currentUser,
    time: +new Date(),
    content: this.newCommentEditor.getEditableContent()
  });
  this.commentsUpdated.next(comments);
  this.newCommentEditor.setEditableContent('');
}
```

There are a few key aspects in this part of the code. This method will be called from our component view when the `Add comment` button is clicked. This is when the user will have

already entered some text into the editor and a new comment will have been created.

First, we will use the user service that we injected within the constructor to obtain information related to the currently logged-in user. The content of the newly created comment will be obtained directly from the `Editor` component we set up using the `@ViewChild` annotation. And, the `getEditableContent` method will allow us to receive the content of the editable element within the in-place editor.

The next thing we wanted to do was to communicate an update of the comment list with the outside world. We used the `commentsUpdated` output property to emit an event with the updated comment list.

Finally, we wanted to clear the editor used to add new comments. As the in-place editor in the view of the `Comments` component is only used to add new comments, we can always clear it after a comment is added. This will give the user the impression that his comment has been moved from the editor into the list of comments. Then, once again, we can access the `Editor` component directly using our `newCommentEditor` property and call the `setEditableContent` method with an empty string to clear the editor. And this is what we've done here.

Our `Comments` component will hold the list of all the comments, and its view will create a `Comment` component for each comment in that list. Each `Comment` component will use an `Editor` component to provide in-place editing of its content. These editors work autonomously using their own controls, and they emit an event if the content is changed or altered in any way. To take care of this, we need to re-emit this event with the name `commentEdited` from the `Comment` component. Now we only need to catch this event within our `Comments` component in order to update the list of comments with the changes. This is illustrated in the following part of the code:

```
onCommentEdited(comment, content) {
  const comments = this.comments.slice();
  if (content.length === 0) {
    comments.splice(comments.indexOf(comment), 1);
  } else {
    comments.splice(comments.indexOf(comment), 1, {
      user: comment.user,
      time: comment.time,
      content
    });
  }
  this.commentsUpdated.next(comments);
}
```

This method will be called for each individual `Comment` component that is repeated using the `NgFor` directive. From the view, we pass a reference to the comment object concerned, as well as the edited content we would receive from the `Comment` component event.

The comment object will only be used to determine the position of the updated comment within the comment list. If the new comment content is empty, we will remove the comment

from the list. Otherwise, we will just create a copy of the previous comment object, change the content with the new edited content, and replace the old comment object in the list with the copy.

Finally, since we wanted to communicate the change in the comment list, we emitted an event using the `commentUpdated` output property.

With this, we have completed our commenting system, and now it's time to make use of it. We already have an empty tab prepared for our project comments, and this is going to be the spot where we will add commenting capabilities using our commenting system.

First, let's amend our `Project` component template, `project/project.html`, to include the commenting system:

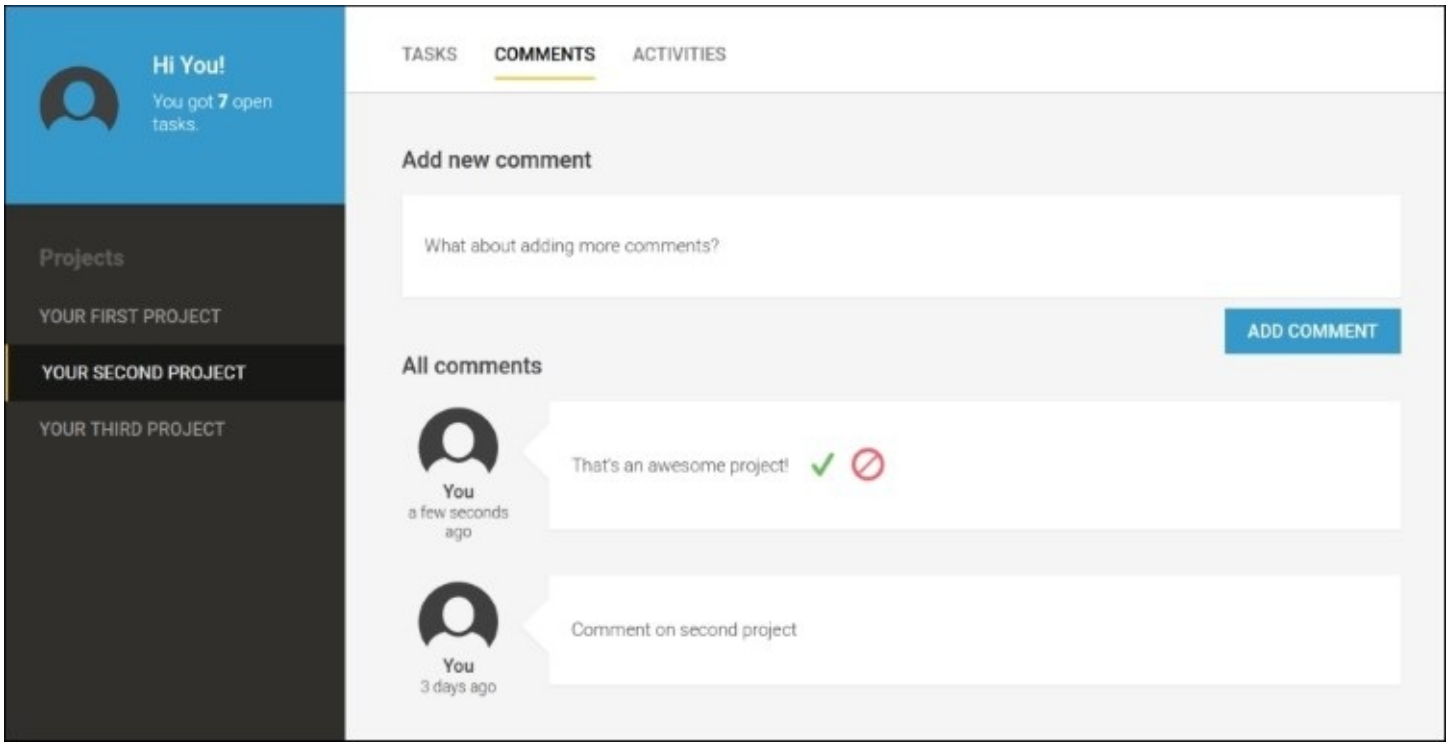
```
...
<ngc-tabs>
  <ngc-tab name="Tasks">...</ngc-tab>
  <ngc-tab name="Comments">
    <ngc-comments [comments]="comments"
                  (commentsUpdated)="updateComments($event)">
    </ngc-comments>
  </ngc-tab>
  <ngc-tab name="Activities"></ngc-tab>
</ngc-tabs>
```

This is as easy as it gets. Since we are paying attention to a clean component architecture, the inclusion of our commenting system really works like a breeze. The only thing we now need to ensure is that we provide a property on our project with a list of comments. We also need a way to react to changes if comments are updated within our `Comments` component. For this purpose, we will create an `updateComments` method.

Let's look at the component class changes within the `project/project.js` file:

```
export class Project {
  ...
  @Input() comments;
  ...
    updateComments(comments) {
      this.projectUpdated.next({
        comments
      });
  }
}
```

Since we are already dealing with project updates in a general way and our `Project` component is emitting directly to our `App` component, where projects data will be persisted, the only thing we need to implement is an additional input property, as well as a method to handle comment updates:



Screenshot of the commenting system integrated within our project component

Recap

Within this topic, we have successfully created a full-fledged commenting system that can be placed in various areas of our application to enable commenting. Users can interact with in-place editors to edit the content in comments, which gives them a great user experience.

While writing the code for our commenting system, we learned about the following topics:

1. Implementing a simple pipe using the `@Pipe` annotation and the `Moment.js` library to provide relative time formatting
2. Using the `OnChanges` life cycle hook to prevent unwanted or invalid values within input properties
3. Using `@ViewChild` to obtain a reference to the components within the component sub-tree in order to establish direct communication
4. Reusing the `Editor` component as an input field replacement and as an autonomous in-place editor within the `Comment` component

Summary

In this chapter, we created a simple in-place editor that we can use for making content editable within our application. Going forward, we can use the `Editor` component wherever we want to make content editable for our users. They will not have to jump into nasty dialogs or separate configuration pages, but will be in a position to edit directly, as per their current context. This is a great tool for enhancing the user experience for our users.

Besides our shiny new `Editor` component, we created a whole commenting system that can be easily included in areas of our application where we'd like to provide commenting capabilities. We have now included the commenting system within our `Project` component, and users can now comment on projects by navigating to the **Comments** tab on the project details.

In the next chapter, we'll use the component-based router of Angular to make our application navigable.

Chapter 5. Component-Based Routing

Routing is an integral part of today's frontend applications. In general, a router serves two main purposes:

- Making your application navigable so that users can use their browser's back button and store and share links within the application
- Offload parts of the application composition so that the router takes responsibility to compose your application, based on routes and route parameters

The router that comes with Angular supports many different use-cases, and it comes with an easy-to-use API. This supports child routers that are similar to the Angular UI-Router nested states, Ember.js nested routes, or child routers in the Durandal framework. Tied to the component tree, this also makes use of its own tree structure to store states and to resolve requested URLs.

In this chapter, we refactor our code to use the component-based router of Angular. We will look into the core elements of the router and how to use them to enable routing in our application.

The following topics will be covered in this chapter:

- An introduction to the Angular router
- An overview of the refactoring to enable the router in our application
- Composition by template, composition by routing, and how to mix them
- Using the `Routes` decorator to configure routes and child routes
- Using the `OnActivate` router lifecycle hook to obtain route parameters
- Using the `RouterOutlet` directive to create insertion points that are controlled by the router
- Using the `RouterLink` directive and the router DSL to create navigation links
- Querying for the `RouterLink` directives using the `@ChildView` decorator in order to obtain the link's active state

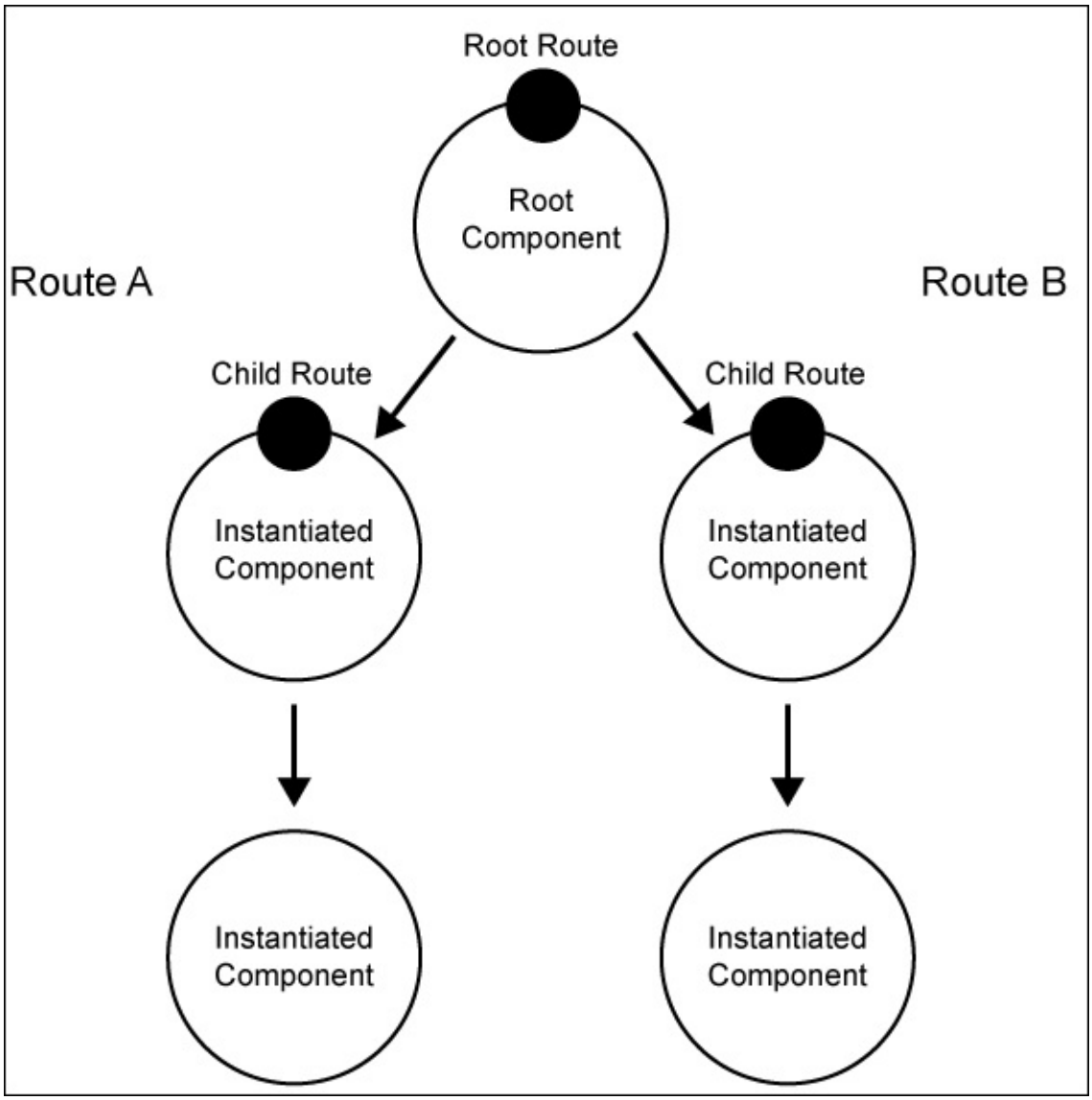
An introduction to the Angular router

The router in Angular is closely coupled to our component tree. The design of the Angular router is built on the assumption that a component tree is directly related to our URL structure. This is certainly true for most of the cases. If we look at a component **B**, which is nested within a component **A**, the URL to represent our location would very likely be `/a/b`.

In order to specify the location in our template where we'd like to enable the router to instantiate components, we can use so-called outlets. Simply by including a `<router-outlet>` element, we can make use of the `RouterOutlet` directive to mark the router insertion point in our template.

Based on some route configuration that we can place on our component, the router then decides which components need to be instantiated and placed into the router outlets. Routes can also be parameterized, and we can access these parameters within the instantiated components.

Based on our component tree and the route configurations on components in this tree, we can build a hierarchical routing and decouple child routes from their parent routes. Such nested routes make it possible to specify route configuration on multiple layers in our component tree and reuse parent components for multiple child routes.



Router hierarchy established through a component tree

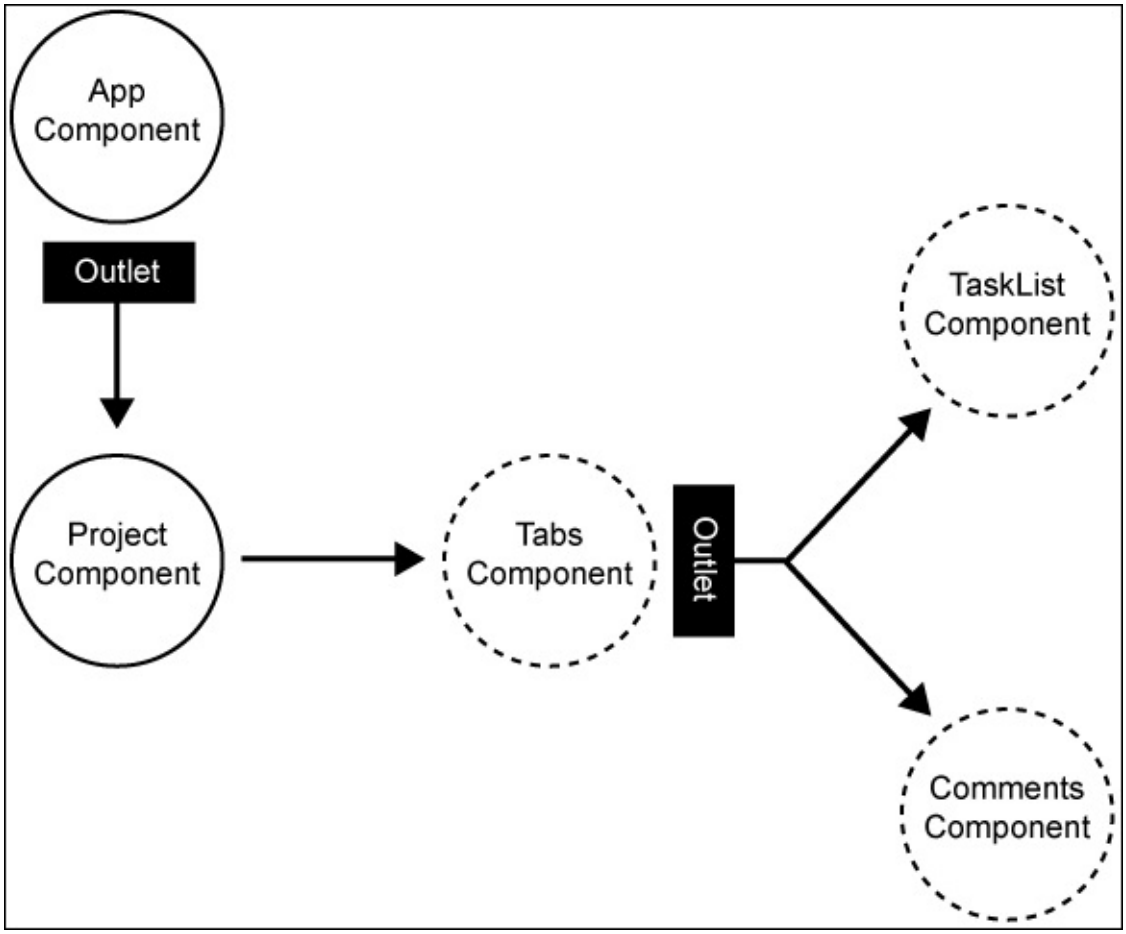
Let's look at the elements of the router again in more detail:

- **Route configuration**: The route configuration is placed at component level, and it contains the different routes possible for this level in the component tree. By placing multiple route configurations on different components in the component tree, we can build decoupled nested routes easily.
- **Router outlets**: Outlets are the locations in components that will be managed by the router. Instantiated components that are based on the route configuration will be placed into these outlets.
- **Router link**: These are links built with a DSL style notation that enable the developer to build complex links through the routing tree.

Composition by routing

So far, we achieved composition by including subcomponents in component templates. However, we'd now like to give the control to the router to decide which component should be included and where.

The following illustration provides an overview of the component architecture of our application, which we're going to enable to route:



A component tree displaying routing components (solid line) and router outlets

The `Project` component is now not directly included with our `App` component. Instead, we use a router outlet in the template of our `App` component. This way, we can give control to the router, and let it decide which component should be placed into the outlet. The `App` component's router configuration will contain all top-level routes. In the current application, we only have the `Project` component as a secondary-level component, but this will change in further chapters.

The `Project` component contains child route configuration to navigate to the tasks and comments view. However, it does not directly contain a router outlet. We use the `Tabs`

component as a navigation element for any sub views. As a result, we'll place the router outlet into the `Tabs` component and include the component directly in the template of the `Project` component.

Router versus template composition

The composition that we dealt with so far was purely based on instantiation via template inclusion. We used input and output properties to decouple and encapsulate components, and followed nice reusable patterns.

With the router, we face a problem that has not yet been solved by Angular and requires that we find our own solution. As we give control to the router to instantiate and insert components into our component tree, we can't control any bindings on our instantiated component. While we previously relied on the clean decoupling of components using input and output properties, we can no longer do this. The only thing that a router provides us are route parameters that may have been set along the activated route.

This puts us in quite a nasty situation. Basically, we need to decide between two designs when writing components, which are as follows:

- We use a given component purely in template composition and, therefore, rely on input and output properties as the glue between the parent component
- We use a component instantiated by the router and rely on input-provided view route parameters and don't require communication with the parent component

Well, both of the preceding design approaches aren't very nice, are they? In an ideal world, we would not need to apply any changes to a component when we enable it for routing. The router should just enable the component for routing, but it should not require any changes on the component itself. Unfortunately, there's no agreement for a solution to this problem at the time of writing this book.

As we don't want to lose any composition capabilities that we gain from relying on inputs and outputs in our `TaskList` and `Comments` components, we need to a find a better solution to enable routing in our application.

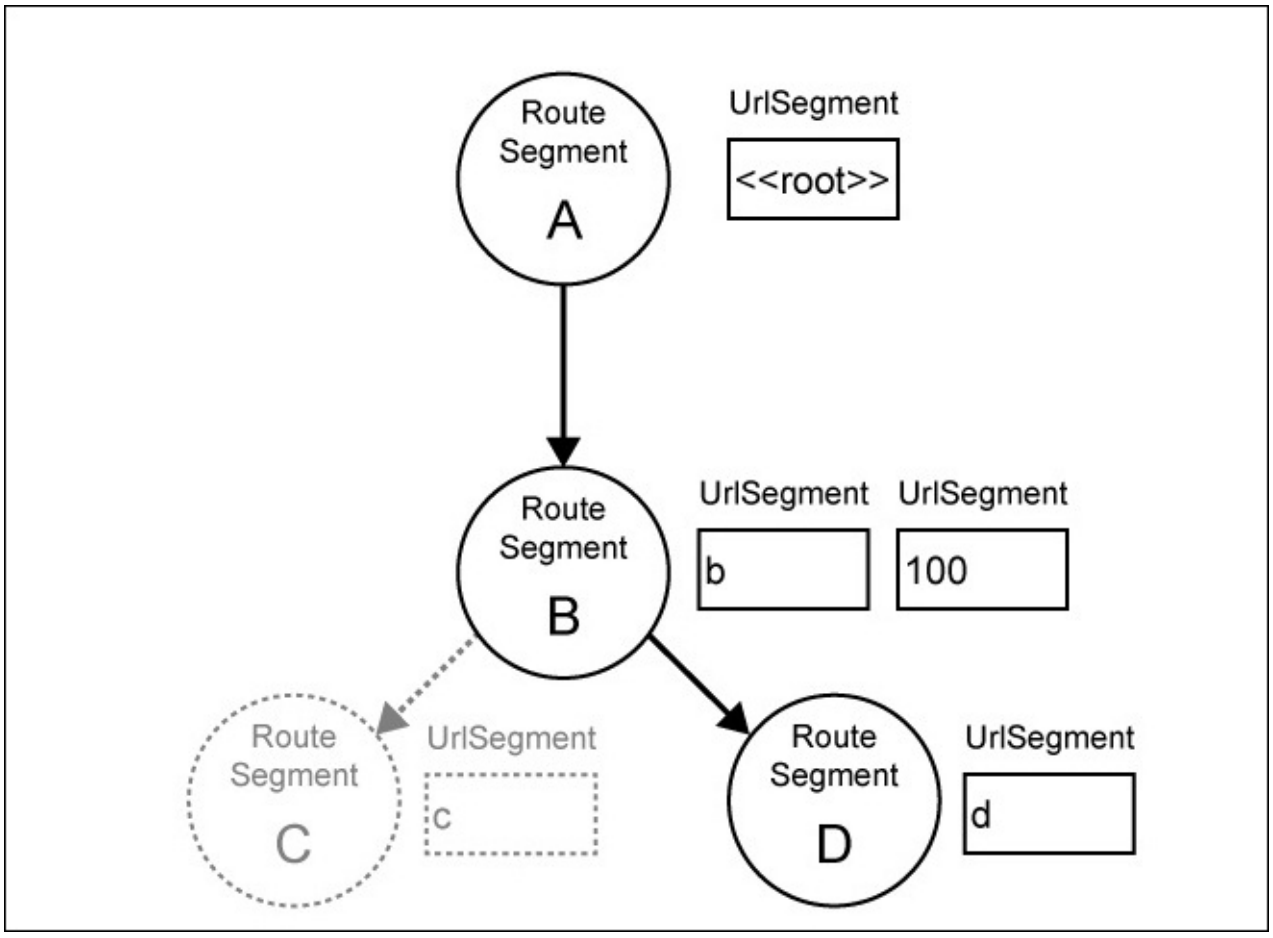
- The following solution allows us to leave the `TaskList` and `Comments` components untouched while they can still rely on input and output properties. Instead of exposing them directly to the router, we will build wrapper components that we address from our routes. These wrappers follow some mechanics to bridge this gap between the router and our components.
- The `wrapper` component deals with any route parameters or route data that might have been set in the activated route.
- Their template should only include the component that is wrapped and its input and output bindings.
- They handle the required data and functionality to provide the input and output bindings of the concerned component
- They may use parent component injection to establish communication with the parent and propagate any action that is required by emitted events. Parent component injection should be used with caution as it somewhat breaks our decoupling of components.

Understanding the route tree

Angular uses tree data structures to represent the router state. You can imagine that every navigation in your application activates a branch in this tree. Let's look at the following example.

We have an application that consists of four possible routes:

- `/`: This is the root route of the application, which is handled in a component called **A**.
- `/b/:id`: This is the route where we can access a `b` detail view, which is handled in a component called **B**. In the URL, we can pass an `id` parameter (that is, `/b/100`).
- `/b/:id/c`: This is the route where the `b` detail view has another navigation possibility, which reveals more specific details that we call `c`. This is handled in a **C** component.
- `/b/:id/d`: This is the route where we can also navigate to a `d` view in the `b` detail view. This is handled by a component, called **D**:



A route tree consisting of an active branch of route segments for an activated route /b/100/d

Let's assume that we activate a route in our example by navigating the URL, `/b/100/d`. In this case, we'd activate a route that reflects the state that is outlined in the preceding figure. Note that the route segment **B** actually consists of two URL segments. The reason for this is that

we've specified that our route **B** actually consists of the `b` identifier and an `:id` route parameter.

Using this tree data structure, we have a perfect abstraction to deal with navigation trees. We can compare trees, check whether certain segments exist in a tree, and extract parameters present on resolved route segments.

To demonstrate the use of routing trees, let's take a look at the `OnActivate` router lifecycle hook that we can implement on our navigable components:

```
routerOnActivate(currentRouteSegment,
                 previousRouteSegment,
                 currentTree,
                 previousTree)
```

When we implement this lifecycle hook on our components, we can run some code after the route was activated. The `currentRouteSegment` argument will point to the `RouteSegment` instance that was activated on our component.

Let's take a look at our example again and assume that we want to access the `:id` parameter in the `routerOnActivate` hook of our **B** component:

```
routerOnActivate(currentRouteSegment) {
  this.id = currentRouteSegment.getParam('id');
}
```

Using the `getParam` function on the `RouteSegment` instance, we obtain any parameters that are resolved on the given segment. In our example case, this would return a `100` string.

Let's take a look at a more complex example. What if we want to access the `:id` parameter from the **D** component on the `d` detail view? In the `OnActivate` lifecycle hook, we'll receive only the route segment that is relevant to the **D** component. This only consists of the `d` URL segment and this does not include the `:id` parameter from the parent route. We can now make use of the `RouteTree` instance to find the parent route segment and obtain the parameter from there:

```
routerOnActivate(currentRouteSegment,
                 previousRouteSegment,
                  currentTree) {
  this.id = currentTree.parent(currentRouteSegment).getParam('id');
}
```

Using the current `RouteTree` instance, we can obtain the parent of the current route segment. As a result, we'll receive the parent route segment (`RouteSegment` **B** in the preceding figure) from where we can obtain the `:id` parameter.

As you can see, the router API is quite flexible, and it allows us to inspect route activity on a very fine granularity. The tree structures that are used in the router make it possible to compare complex router states in our application without bothering about the underlying

complexity.

Back to the routes

All right, now it's time to implement routing for our application! In the following topics, we'll create the following routes for our application:

Route path	Description
`/projects/:projectId`	This route will activate the `Project` component in the outlet of our main application component. This consists of the `projects` URL segment as well as the `:projectId` URL segment to specify the project ID.
`/projects/:projectId/tasks`	This route will activate the `TaskList` component. We will create a `ProjectTaskList` wrapper component in order to decouple our `TaskList` component from routing. We'll apply the procedure described in the previous section, *Router versus template composition*.
`/projects/:projectId/comments`	This route will activate the `Comments` component. We'll create a `ProjectComments` wrapper component in order to decouple our `Comments` component from routing. We'll apply the procedure described in the previous section, *Router versus template composition*.

In order to use the router of Angular, the first thing that we need to do is to add the route provider to our application. We'll do this on bootstrap in order to make sure the router providers are only loaded once. Let's open our `boostrap.js` file and add the necessary dependencies:

```
...
import {bootstrap} from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';
// Import router dependencies
import {HashLocationStrategy, LocationStrategy} from '@angular/common';
import {ROUTER_PROVIDERS} from '@angular/router';
...
bootstrap(App, [
  ...
  ROUTER_PROVIDERS,
  provide(LocationStrategy, {
    useClass: HashLocationStrategy
  })
]);
```

From the router module, we import the `ROUTER_PROVIDERS` constant that contains a list of modules that are required to be exposed as providers when using the router. We also import the `LocationStrategy` and `HashLocationStrategy` type from the common module that need to be provided manually.

Using the `provide` function, we provide the `HashLocationStrategy` class as a substitution for the `LocationStrategy` abstract class. This way, the router will know which strategy to use when resolving URLs.

The following two strategies exist at the moment:

- **HashLocationStrategy**: This can be used when the router should use hash URLs, such as `localhost:8080#/child/something`. This location strategy makes sense if you're working in an environment where the HTML5 push state can't be used due to browser or server constraints. The whole navigation state will be managed in the fragment identifier of the URL.
- **PathLocationStrategy**: This strategy can be used if you'd like to use the HTML5 push state to handle application URLs. This means that your application navigation becomes the actual path of the URL. Using the preceding example of a hash-based URL, this strategy would enable the direct use of `localhost:8080/child/something`. As the initial requests will hit the server if the state is encoded in the path of a URL, you'll need to enable the correct routing on the server to make this work properly.

After enabling the router for our application, we will need to make our root component routable. We can do this by including a route configuration on our `App` component. Let's look at the necessary code to do this. We edit the `app.js` file in our `lib` folder:

```
...
import {Project} from './project/project';
import {Routes, Route} from '@angular/router';

@Component({
  selector: 'ngc-app',
  ...
})
@Routes([
  new Route({path: 'projects/:projectId', component: Project})
])
export class App {
     ...
}
```

In the preceding code, we imported the `Routes` decorator as well as the `Route` type from the router module.

In order to configure routes on our component, we can use the `@Routes` decorator by passing an array of `Route` objects that describe the possible child routes on this component.

Let's look at the available options that we can pass to the `Route` constructor:

Route property	Description

`path`	This property is required. Using the path, we can describe the navigation URL in the browser using the route matcher DSL. This can contain route parameter placeholders.
	
	Some examples are as follows:
	• The following route gets activated if the user navigates to `/home` in the browser:
	`path: '/home'`
	• The following route gets activated when the user navigates to a `/child/something` URL where something will be available as route parameter with the name, `id`:
	`path: '/child/:id'`

`component`	This property is required, and it defines which component should be instantiated by the router. As already explained in the previous section, the router does not allow us here to specify any bindings to the instantiated component.

The route configuration on our `App` component covers the `Project` component being instantiated on the `projects/:projectId` route path. This means that we use a `projectId` parameter on the child route, which will be available to the `Project` component.

We also need to modify our `App` component template and remove the direct inclusion of the `Project` component there. We now give control to the router to decide which component to display. For this, we need to make use of the `RouterOutlet` directive to provide a slot in our template where the router will instantiate components.

The `RouterOutlet` directive is part of the `ROUTER_DIRECTIVES` constant that is exported by the router module. Let's import and add the constant to the directives list on our component:

```
...
import {Routes, Route, ROUTER_DIRECTIVES} from '@angular/router';
   ...
@Component({
  selector: 'ngc-app',
  ...
  directives: [..., ROUTER_DIRECTIVES],
  ...
})
...
```

```
export class App {
  ...
}
```

Now, we can use the `RouterOutlet` directive in our template to indicate the insertion position of instantiated components by the router. Let's open our `App` component template file, `app.html`, and make the necessary modifications:

```
<div class="app">
  ...
  <div class="app__l-main">
    <router-outlet></router-outlet>
  </div>
</div>
```

The next step is to refactor our `Project` component so that it can be used in routing. As we already outlined in the previous section, the router comes with certain constraints when it comes to component design. For the `Project` component, we decide to redesign it in a way so that we can only use it with routing. This isn't a bad thing here because we can exclude the possibility that it will be reused somewhere else in our application.

The redesign of the `Project` component includes the following steps:

1. Getting rid of all input and output properties of the component.
2. Using the `OnActivate` router lifecycle hook to obtain the `projectId` parameter from the activated route segment of the `App` component.
3. Obtaining the project data directly from the data store using the `projectId` parameter.
4. Handling updates on the project data directly on the component instead of delegating to the `App` component.

Let's modify the `Component` class located in `lib/project/project.js` to implement the preceding design changes:

```
import {Component, ViewEncapsulation, Inject} from '@angular/core';
import template from './project.html!text';
import {Tabs} from '../ui/tabs/tabs';
import {DataProvider} from '../../data-access/data-provider';
import {LiveDocument} from '../../data-access/live-document';

@Component({
  selector: 'ngc-project',
  host: {class: 'project'},
  template,
  encapsulation: ViewEncapsulation.None,
  directives: [Tabs]
})
export class Project {
  constructor(@Inject(DataProvider) dataProvider) {
    this.dataProvider = dataProvider;
    this.tabItems = [
      {title: 'Tasks', link: ['tasks']},
```

```
      {title: 'Comments', link: ['comments']}
    ];
  }

  routerOnActivate(currentRouteSegment) {
    this.id = currentRouteSegment.getParam('projectId');
    this.document = new LiveDocument(this.dataProvider, {
      type: 'project',
      _id: this.id
    });
    this.document.change.subscribe((data) => {
      this.title = data.title;
      this.description = data.description;
      this.tasks = data.tasks;
      this.comments = data.comments;
    });
  }

  updateTasks(tasks) {
    this.document.data.tasks = tasks;
    this.document.persist();
  }

  updateComments(comments) {
    this.document.data.comments = comments;
    this.document.persist();
  }

  ngOnDestroy() {
    this.document.unsubscribe();
  }
}
```

Besides implementing these changes that are already described in the redesigning steps, we also made use of a new `LiveDocument` utility class that we imported from the `data-access` folder. This helps us to keep our programming reactive when we're concerned about changes on a single data entity. Using the `LiveDocument` class, we can query the database for a single entity, while the change property of the `LiveDocument` instance is an observable that keeps us notified about changes on the entity. A `LiveDocument` instance also exposes the data of the entity into a `data` property, which can be accessed directly. If we'd like to make an update on the entity, we can add, modify, or remove properties on the data object and then store the changes by calling `persist()`.

As our `Project` component is now activated by the router in the `App` component, we can make use of the `OnActivate` router lifecycle hook by implementing a method, named `routerOnActivate`. We use the `getParam` function of the current route segment to obtain the `:projectId` parameter of the route.

In the `subscribe` function on the change observable of our `LiveDocument` instance, we expose the project data directly on the `Project` component. This simplifies later use in the view.

In the `OnDestroy` lifecycle hook, we make sure that we unsubscribe from the document change

observable.

Now, we can rely on the `projectId` route parameter to be passed into our component, which makes the `Project` component depend on the router. We got rid of all input properties, and then we set the necessary data by querying our data store using the project ID.

Now, it's time to build the wrapper components that we talked about in order to route to our `TaskList` and `Comments` components.

Let's create a new component called `ProjectTaskList`, which will serve as a wrapper to enable the `TaskList` component in routing. We will create a `project-task-list.js` file in the `lib/project/project-task-list` path, as follows:

```
import {Component, ViewEncapsulation, Inject, forwardRef} from '@angular/core';
import template from './project-task-list.html!text';
import {TaskList} from '../../task-list/task-list';
import {Project} from '../project';

@Component({
  selector: 'ngc-project-task-list',
  ...
  directives: [TaskList]
})
export class ProjectTaskList {
  constructor(@Inject(forwardRef(() => Project)) project) {
    this.project = project;
  }

  updateTasks(tasks) {
    this.project.updateTasks(tasks);
  }
}
```

Let's also take a look at the template in the `project-task-list.html` file:

```
<ngc-task-list [tasks]="project.tasks"
               (tasksUpdated)="updateTasks($event)"></ngc-task-list>
```

We inject the `Project` parent component into our wrapper component. As we can't rely on output properties any more to emit events, this is the only way to communicate with the parent `Project` component. We're dealing with a circular reference here (`Project` depends on `ProjectTaskList`, and `ProjectTaskList` depends on `Project`), hence we need to use a `forwardRef` helper function to prevent the `Project` type evaluating to `undefined`.

If we receive a `tasksUpdated` event in the template, we will call the `updateTasks` method on our wrapper component. The wrapper then simply delegates the call to the project component.

Similarly, we use the project data to obtain the list of tasks and create a binding to the `tasks` input property of the `TaskList` component.

Using this wrapper approach for routing, we're able to leave our components unmodified when enabling them for routing. This is much better than the option to make our task list only available for the router. We would lose the freedom to use a task list outside of the context of a project, and then use it with pure template composition.

For the `Comments` component, we perform the exact same task, and create a wrapper on the `lib/project/project-comments` path. Besides dealing with comments instead of tasks, the code looks exactly the same as with the `ProjectTaskList` wrapper component.

After creating the two wrapper components, we can now create the router configuration on our `Project` component. Let's modify the `project/project.js` file to enable routing:

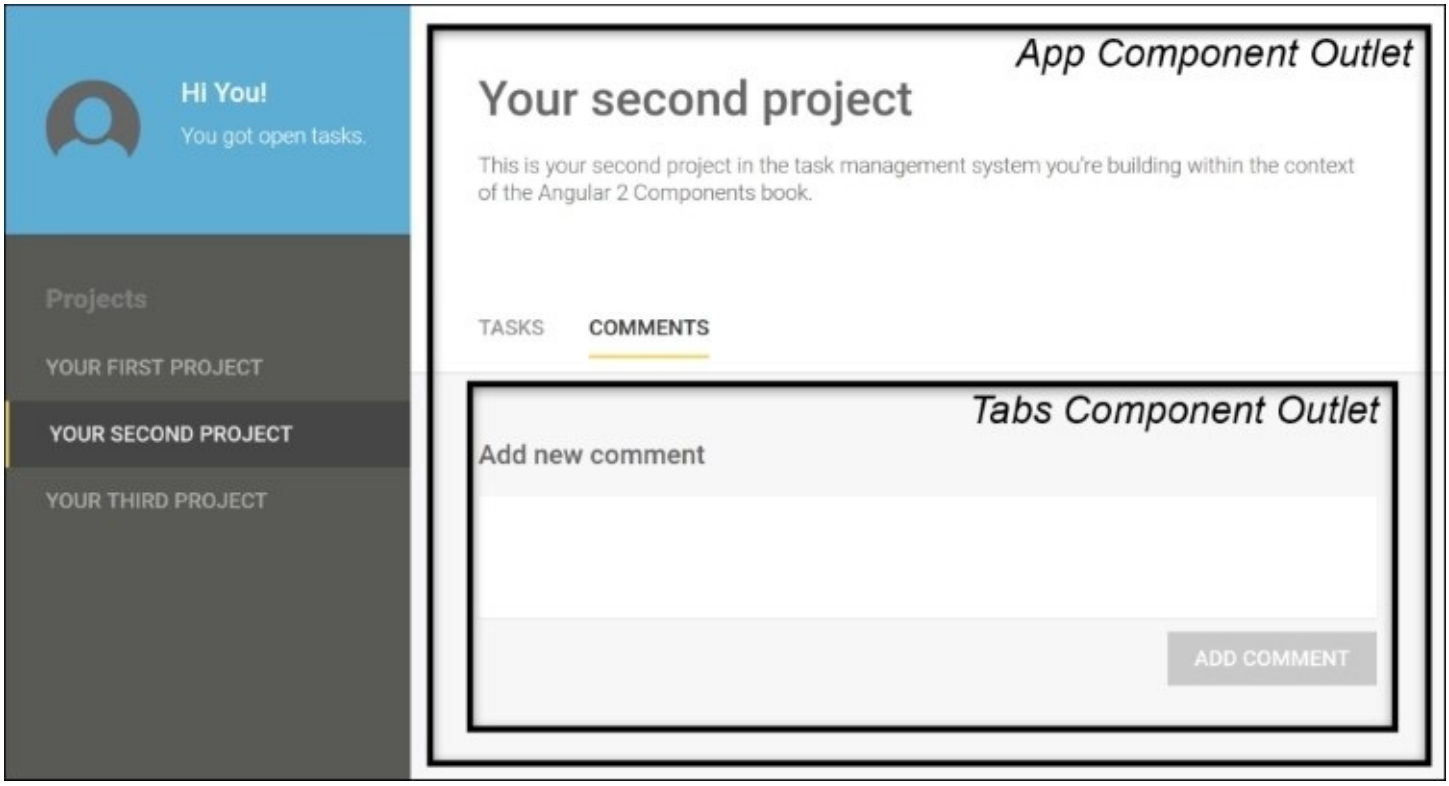
```
...
import {ProjectTaskList} from './project-task-list/project-task-list';
import {ProjectComments} from './project-comments/project-comments';
import {Routes, Route} from '@angular/router';
...
@Component({
  selector: 'ngc-project',
  ...
})
@Routes([
  new Route({ path: 'tasks', component: ProjectTaskList}),
  new Route({ path: 'comments', component: ProjectComments})
])
export class Project {
  ...
}
```

To enable the task list and make comments navigable using the router, we simply create a router configuration that instantiates our wrapper components. We also specify that the tasks route should be the default route if no child route was selected.

Routable tabs

Okay, if you've read through this chapter so far, you now may wonder where the router will instantiate the components of the child routes. We've not yet included a router outlet in the `Project` component template so that the router knows where to instantiate components.

We won't include the outlet for the project router directly in the `Project` component. Instead, we will use our `Tabs` component to take over this job. Instead of using content insertion in our `Tabs` component like we did so far, we now use a router outlet to compose its content. This will make our `Tabs` component unusable for nonrouting cases, but we can establish a nice decoupling by only providing the router outlet. This way we can still reuse the `Tabs` component in other routing situations:



The App component includes a router outlet directly; however, the Project component relies on the Tabs component to provide a router outlet.

On a higher level, we can describe the new design of our `Tabs` component, as follows:

- It renders all tab buttons, based on a list of router links and titles to provide a router navigation
- It provides a router outlet that will be used by the parent component to instantiate navigated components

Let's modify our `Tabs` component in `lib/ui/tabs/tabs.js` to implement these changes:

```
...
import {ROUTER_DIRECTIVES} from '@angular/router';

@Component({
  selector: 'ngc-tabs',
  ...
  directives: [ROUTER_DIRECTIVES]
})
export class Tabs {
  @Input() items;
}
```

The `ROUTER_DIRECTIVES` constant from the router module contains the `RouterOutlet` directive as well as the `RouterLink` directive. By importing the constant and providing it to the components directives list, we enable both router directives to be used in our template.

The `RouterOutlet` directive is used inside the `Tabs` component template to indicate the instantiation point for the router.

The `RouterLink` directive can be used to generate routing URLs from the template using the router link DSL. This allows you to generate navigation links in your application and it can be placed both on anchor tags as well as other elements where it will trigger navigation on click.

The `items` input is an array of link items that contain a title and a router link. On our parent project component, we already prepare these items in the constructor.

Let's also take a quick look at the template of our component in the `tabs.html` file:

```
<ul class="tabs__tab-list">
  <li *ngFor="let item of items">
    <a class="tabs__tab-button"
       [routerLink]="item.link">{{item.title}}</a>
  </li>
</ul>
<div class="tabs__l-container">
  <div class="tabs__tab tabs__tab--active">
    <router-outlet></router-outlet>
  </div>
</div>
```

As we let the router deal with the active view using a router outlet, there's no need any more to use multiple tab components that are switched active. We will always have one active tab, and let the router handle the content.

Let's see how we can make use of the new `Tabs` component in our `Project` component to make the configured routes navigable. First, we need to add the following code to our `Project` component constructor to provide the necessary navigation items to our `Tabs` component:

```
this.tabItems = [
  {title: 'Tasks', link: ['tasks']},
  {title: 'Comments', link: ['comments']}
```

```
];
```

In the link property of our navigation items, we use the router link DSL to specify which route should be navigated. As the navigation is relative to the parent route segment and we're already in the `/projects/:projectId` route, the only thing in our router link DSL should be a relative path to the `tasks` and `comments` child routes.

In the template of our `Project` component, we can now use the `tabItems` property to create a binding to the input property of the `Tabs` component:

```
<div class="project__l-header">
  <h2 class="project__title">{{title}}</h2>
  <p>{{description}}</p>
</div>
<ngc-tabs [items]="tabItems"></ngc-tabs>
```

Refactoring navigation

As a final step, we also need to refactor our navigation components to rely on the router. So far, we used our own routing that was implemented in a complex nested-navigation component structure. We can simplify this a lot using the Angular router.

Let's start with the smallest component first, and edit our `NavigationItem` component template in the `lib/navigation/navigation-section/navigation-item/navigation-item.html` file:

```
<a class="navigation-section__link"
   [class.navigation-section__link--active]="isActive()"
   [routerLink]="link">{{title}}</a>
```

Instead of controlling the link behavior ourselves, we now use the `RouterLink` directive to generate a link that is based on the component `link` input property. To set the active class on the navigation link, we still rely on the `isActive` method on our component, and there's no change required in the template.

Let's look at the changes to the `Component` class in the `navigation-item.js` file:

```
...
import {RouterLink} from '@angular/router/src/directives/router_link';

@Component({
  selector: 'ngc-navigation-item',
  ...
  directives: [RouterLink]
})
export class NavigationItem {
  @Input() title;
  @Input() link;
  @ViewChild(RouterLink) routerLink;

  isActive() {
    return this.routerLink ?
                    this.routerLink.isActive : false;
  }
}
```

Instead of relying on the `Navigation` component to manage the active state of navigation items, we now rely on the `RouterLink` directive. Each `RouterLink` directive provides an accessor property, `isActive`, which tells us whether this specific route addressed by the router link is currently activated within the browsers URL. Using the `@ViewChild` decorator, we can query for the `RouterLink` directive in our view and then query the `isActive` property to find out if the current navigation item is active or not.

Now, we only need to make sure that we pass the necessary items to the `Navigation` component in our `App` component in order to make our navigation work again.

The following code needs to be changed in the `App` component constructor in the `app.js` file:

```
this.projectsSubscription = projectService.change
  .subscribe((projects) => {
    this.projects = projects;
    // We create new navigation items for our projects
    this.projectNavigationItems = this.projects
      // We first filter for projects that are not deleted
      .filter((project) => !project.deleted)
      .map((project) => {
        return {
          title: project.title,
          link: ['/projects', project._id]
        };
      });
    });
```

By filtering and mapping the available projects, we can create a list of navigation items that contain a `title` and `link` property. The `link` property contains a route link DSL that points to the project details route that is configured in the `App` component router configuration. By passing an object literal as a sibling to the route name, we can specify some parameters along the route. Here, we simply set the expected `projectId` parameter to the ID of the project in the projects list.

Now, our navigation components make use of the router to enable navigation. We got rid of our custom routing functionality in the `Navigation` component, and we use router link DSL to create navigation items.

Summary

In this chapter, we learned about the basic concepts of the router in Angular. We looked at how we can use the existing component tree to configure child routes in nested-router scenarios. Using nested-child routes, we enabled the reuse of components with route configurations.

We also looked at the problem of router versus template composition and how to mitigate this problem using wrapper components. In this way, we close the gap between the router and underlying components using an in-between layer.

We looked into route configuration specifics and the basics of the router link DSL. We also covered the basics of the `RouteTree` and `RouteSegment` classes and how to use them to perform in-depth route analysis.

In the next chapter, we will learn about SVG and how to use this web standard in order to draw graphics in our Angular applications. We will visualize an activity log of our application activities using SVG and see how Angular makes this technology even greater by enabling composability.

Chapter 6. Keeping Up with Activities

In this chapter, we'll build an activity log in our task management system using **Scalable Vector Graphics** (**SVG**) to build graphical components using Angular. SVG is the perfect candidate when it comes to complex graphical content, and using Angular components, we can build nicely encapsulated and reusable content.

Since we want to log all the activities within our application, such as adding comments or renaming tasks, we are going to create a central repository. We can then display these activities and render them as an activity timeline using SVG.

To add an overview of all the activities and to provide a user input to narrow the range of displayed activities, we're going to create an interactive slider component. This component will use a projection to render timestamps, in the form of ticks and activities, directly onto the slider's background. We'll also use SVG to render the elements within the component.

We'll cover the following topics in this chapter:

- A basic introduction to SVG
- Making SVG composable with Angular components
- Using namespaces in component templates
- Creating a simple pipe to format calendar times using Moment.js
- Using the `@HostListener` annotations to handle user input events to create an interactive slider element
- Making use of Shadow DOM using `ViewEncapsulation.Native` in order to create native-style encapsulation

Creating a service for logging activities

The goal of this chapter is to provide a way to keep track of all user activities within the task management application. For this purpose, we'll need a system that will allow us to log activities within components and to access already logged activities.

Activities, as entities, should be quite generic and should have the following fields with their respective purposes:

- **Subject**: This field should be used to reference the subject of the activity. This can be any identifier that identifies a foreign entity. In the context of projects, we'll store the project ID in this field. Services and components that use the activity service should use this field to filter specific activities further.
- **Category**: This field provides an additional way of tagging the activity further. For projects, we will currently use two categories: *comments* and *tasks*.
- **Title**: This refers to the title of the activity that will provide a very brief summary of what the activity is about. This could be something like *New task was added* or *Comment was deleted*.
- **Message**: This is the field where the real beef of the activity goes into. It should contain enough information to provide good traceability of what happened during the activity.

In order to develop our system, we'll create a new file named `activity-service.js` under the `activities/activity-service` path in our `lib` folder. In this file, we will create our activity service class, which we're enabling for dependency injection, by using the `@Injectable` annotation:

```
@Injectable()
constructor(@Inject(DataProvider) dataProvider,
                    @Inject(UserService) userService) {
export class ActivityService {
    // We're exposing a replay subject that will emit events
    // whenever the activities list change
    this.change = new ReplaySubject(1);
    this.dataProvider = dataProvider;
    this.userService = userService;
    this.activities = [];

    // We're creating a subscription to our datastore to get
    // updates on activities
    this.activitiesSubscription = this.dataProvider.getLiveChanges()
      .map((change) => change.doc)
      .filter((document) => document.type === 'activity')
      .subscribe((changedActivity) => {
        this.activities = this.activities.slice();
        // Since activities can only be added we can assume that
        // this change is a new activity
        this.activities.push(changedActivity);
        // Sorting the activities by time to make sure there's no
        // sync issue messing with the ordering
        this.activities.sort((a, b) =>
```

```
          a.time > b.time ? -1 : a.time < b.time ? 1 : 0);
        this.change.next(this.activities);
      });
  }

  // This method is logging a new activity
  logActivity(subject, category, title, message) {
    // Using the DataProvider to create a new document in our
    // datastore
    this.dataProvider.createOrUpdateDocument({
      type: 'activity',
      user: this.userService.currentUser,
      time: new Date().getTime(),
      subject,
      category,
      title,
      message
    });
  }
}
```

In the constructor of our activity service, we've subscribed to changes to our data store and have filtered any incoming change by type so we will only receive activity updates.

Since activities can't be edited or deleted, we only need to be concerned about newly added activities. We update the internal array of activities with any added activity in the subscription. This way, we'll not only receive all the initial activities, but also the activities that are subsequently added directly from the data store. Other services and components can then directly access the activity list of the system.

In order for other application components to react to changes in the activity list, we've exposed a `ReplaySubject` observable on the `change` member field.

In the `logActivity` method, we've simply added a new activity to the data store. `UserService` will provide us with information on the currently logged-in user, and we can use `DataProvider` to write to the data store.

So, we have created a simple platform that will help us keep track of activities within our application. Since we want only one instance of `ActivityService` within our application, let's add it to the `providers` list on our root `App` component. You'll find this component in the `app.js` file, located within our `lib` folder:

```
@Component({
  selector: 'ngc-app',
  …
  providers: [ProjectService, UserService, ActivityService]
})
```

Because all dependency injectors will inherit the dependencies from our `App` component, we can inject it in any component of our application going forward.

Logging activities

We have created a nice system to log activities. Now let's go ahead and use it within our components to keep an audit of all the activities.

First, let's use `ActivityService` to log activities within the `TaskList` component. The following code excerpt highlights the changes made to the `TaskList` component within the `task-list/task-list.js` file in our `lib` folder:

```
...
import {ActivityService} from '../activities/activity-service/activity-
service';
import {limitWithEllipsis} from '../utilities/string-utilities';

@Component({
  selector: 'ngc-task-list',
  ...
})
export class TaskList {
  …
  // Subject for logging activities
  @Input() activitySubject;

  onTaskUpdated(task, updatedData) {
    ...
    // Creating an activity log for the updated task
    this.activityService.logActivity(
      this.activitySubject.id,
      'tasks',
      'A task was updated',
      'The task "${limitWithEllipsis(oldTask.title, 30)}" was updated on
#${this.activitySubject.document.data._id}.'
    );
  }

  onTaskDeleted(task) {
    ...
    // Creating an activity log for the deleted task
    this.activityService.logActivity(
      this.activitySubject.id,
      'tasks',
      'A task was deleted',
      'The task "${limitWithEllipsis(removed.title, 30)}" was deleted from
#${this.activitySubject.document.data._id}.'
    );
  }

  addTask(title) {
    ...
    // Creating an activity log for the added task
    this.activityService.logActivity(
      this.activitySubject.id,
      'tasks',
      'A task was added',
```

```
      'A new task "${limitWithEllipsis(title, 30)}" was added to
#${this.activitySubject.document.data._id}.'
    );
  }
  ...
}
```

Using the `logActivity` method of `ActivityService`, we can easily log any number of activities within the already existing `TaskList` methods to modify tasks.

In the message body of our activities, we've used a new utility function, `limitWithEllipsis`, which we've imported from a new module, namely `string-utilities`. This function takes a string and a number as parameters. The returned string is a truncated version of the input string, which is cut off at the position specified with the second parameter. In addition, there's an ellipsis character (. . .) appended to the string. I won't bother you with the rather simple code within this helper. If you'd like to know how it's implemented, you can always check the implementation after downloading this chapter's code.

If you go back to the specification of our activity logs, you will see that we always need to specify a subject in order to log activities. We've implemented this on our `TaskList` component by introducing a new input parameter called `activitySubject`. The assumption here is that each activity subject contains `LiveDocument` stored under the document member. From there, we can obtain the ID in the data store and use it for our activity message.

If we revisit our `Project` component, you will see that we're already following the prerequisites of being an activity subject. We've stored a reference to the underlying `LiveDocument` instance under the `document` member field.

All we need to do now is change the template of our `ProjectTaskList` wrapper component to pass the `activitySubject` project input of the `TaskList` component. Let's look at the changes in the `lib/project/project-task-list/project-task-list.html` file quickly:

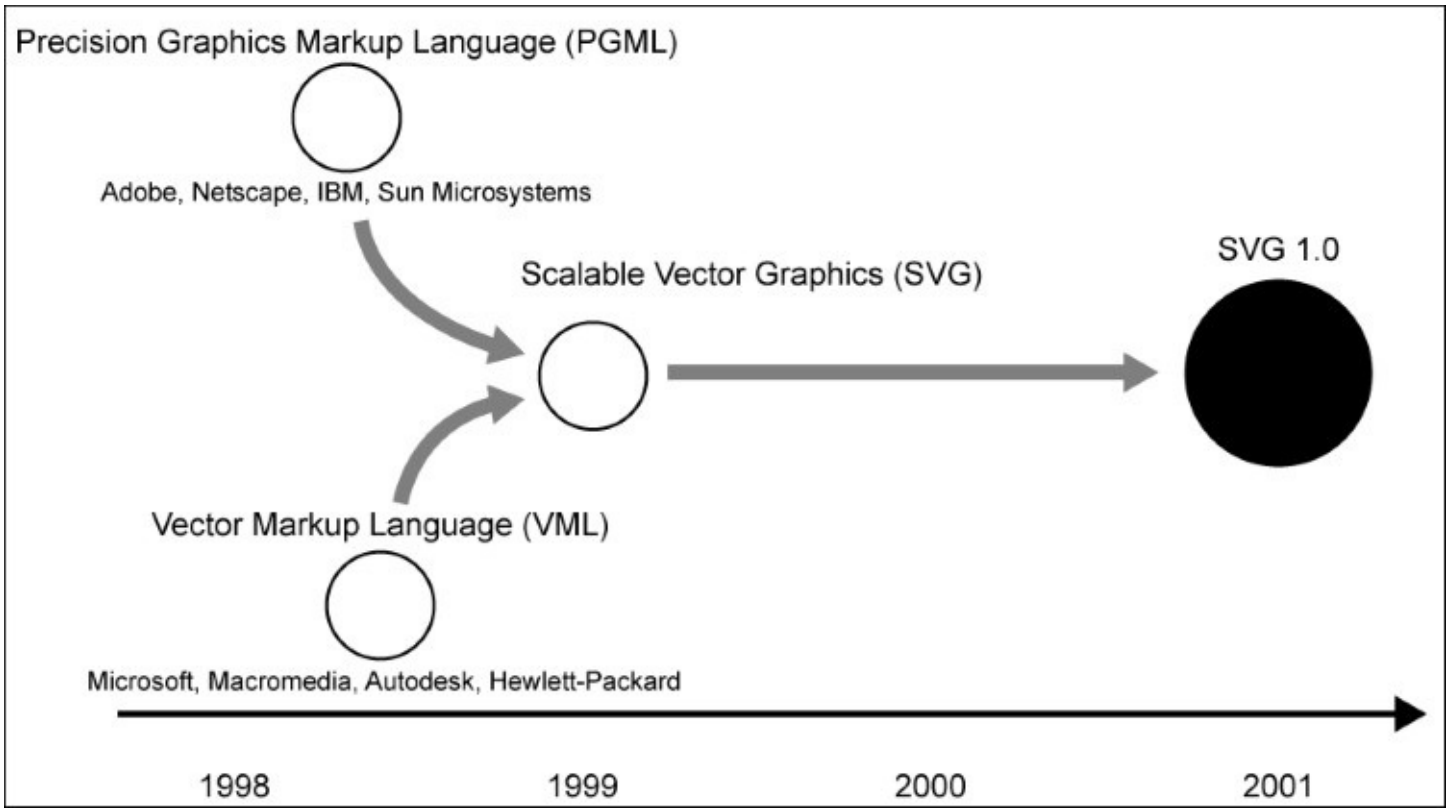
```
<ngc-task-list [tasks]="project.tasks"
               [activitySubject]="project"
             (tasksUpdated)="updateTasks($event)"></ngc-task-list>
```

You might wonder why we care about this rather cumbersome way of dealing with our task list, if we could just pass in a hard reference to the project and use project tasks and the project ID directly. The beautiful aspect of our current solution is that we do not have any dependency on a project as such. We could also use our `TaskList` component without the context of a project. And we can still pass a list of tasks to the `tasks` input and use a different activity subject for the activity logs.

We're also going to use `ActivityService` within the `Comments` component to create logs for added, edited, and deleted comments. Since the steps involved are very similar to what we've just done for the `TaskList` component, we're going to skip this. You can always take a look at the final codebase for this chapter to add activity logs for the `Comments` component.

Leveraging the power of SVG

SVG has been a part of the Open Web Platform standards since 1999 and was first recommended in 2001 under the SVG 1.0 standard. SVG is a consolidation of two independent proposals for an XML-based vector image format. **Precision Graphics Markup Language** (**PGML**)—mainly developed by Adobe and Netscape—as well as **Vector Markup Language** (**VML**)—which was mainly represented by Microsoft and Macromedia—were both different XML formats that served the same purpose. The W3C consortium declined both the proposals in favor of the newly developed SVG standard that unified the best of both worlds into a single standard:



Timeline showing the development of the SVG standard

All three standards had a common goal, which was to provide a format for the Web to display vector graphics in the browser. SVG is a declarative language that specifies graphical objects using XML elements and attributes.

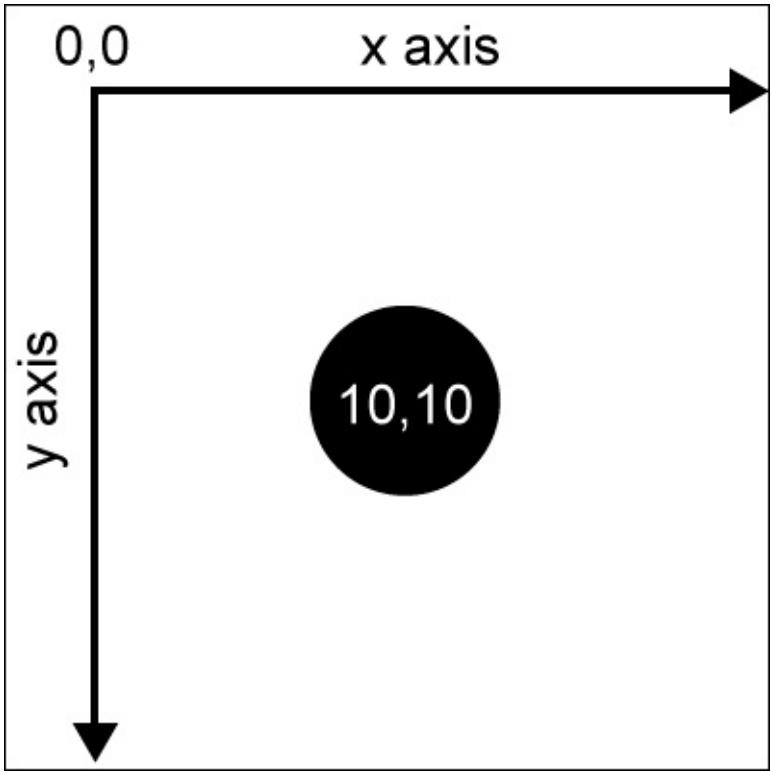
Let's look at a simple example on how to create an SVG image with a black circle, using SVG:

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
     width="20px" height="20px">
  <circle cx="10" cy="10" r="10" fill="black" />
```

```
</svg>
```

This rather simple example represents an SVG image with a black circle, whose center is located at *x = 10 px* and *y = 10 px*. The radius of the circle is 10 px, which makes this circle 20 px in width and height.

The origin of the coordinate system in SVG sits on the top-left corner, where the *y* axis faces the south direction and the *x* axis eastward:



The coordinate system within SVG

Using not only primitive shapes, such as circles, lines, and rectangles, but also complex polygons, the possibilities for creating graphical content are nearly unlimited.

SVG is not only used within the Web, but has also become a very important intermediate format for exchanging vector graphics between different applications. Almost any application that supports vector graphics also supports the import of SVG files.

The real power of SVG comes to the surface when we do not include an SVG file as an HTML image, but rather include the SVG content directly within our DOM. Since HTML5 directly supports the SVG namespace within an HTML document and will render the graphics we define within our HTML, a whole bunch of new possibilities spring up. We can now style our SVG with CSS, manipulate the DOM with JavaScript, and easily make our SVG interactive.

Taking the previous example of our circle image to the next level, we could make it interactive by changing the circle color by clicking it. First, let's create a minimal HTML document and include our SVG elements directly within the DOM:

```
<!doctype html>
<title>Minimalistic Circle</title>
<svg width="20px" height="20px">
  <circle id="circle" cx="10" cy="10" r="10" fill="black">
</svg>
<script>
  document
    .getElementById('circle')
    .addEventListener('click', function(event) {
      event.target.setAttribute('fill', 'red');
    });
</script>
```

As you can see, we can get rid of the version and the XML namespace declaration when we use SVG directly within the DOM of our HTML document. What's interesting here is that we can treat SVG very much like regular HTML. We can assign an ID and even classes to SVG elements and access them from JavaScript.

Within the `script` tag of our HTML document, we can directly access our `circle` element using the ID we've previously assigned to it. We can add event listeners, the way we already know, from regular HTML elements. In this example, we added a `click` event listener and changed the color of our circle to red.

For the sake of simplicity, we used an inline `script` tag in this example. It would of course be much cleaner to have a separate JavaScript file to do the scripting.

Styling SVG

I'm a purist when it comes to the separation of concerns within the Web. I still strongly believe in the separation of structure (HTML), appearance (CSS), and behavior (JavaScript), as well as producing the most maintainable applications when following this practice.

First, it seems weird to have SVG in your HTML, and you might think that this breaks the contract of a clean separation. Why is this graphical content, consisting of only appearance-relevant data, sitting in my HTML that is supposed to only contain raw information? After dealing with a lot of SVGs within a DOM, I have come to the conclusion that we can establish a clean separation when using SVG by dividing our appearance responsibilities into the following two subgroups:

- **Graphical structure**: This subgroup deals with the process of defining the basic structure of your graphical content. This is about shapes and layout.
- **Visual appearance**: This subgroup deals with the process of defining the look and feel of our graphical structures, such as colors, line widths, line styles, and text alignment.

If we separate the concerns of SVG into these groups, we can actually gain great maintainability. Graphical structure is defined by the SVG shapes themselves. They are directly written within our HTML but don't have a particular look and feel. We only store the basic structural information within HTML.

Luckily, all the properties of visual appearance, such as colors, cannot only be expressed through the attributes in our SVG elements; however, there's a corresponding CSS property that allows us to offload all the look-and-feel-relevant aspects of the structure to CSS.

Go back to the example where we drew a black circle; we'll tweak this a bit to fit our demands of separation of concerns so that we can distinguish graphical structure from graphical appearance:

```
<!doctype html>
<title>Minimalistic Circle</title>
<svg width="20px" height="20px">
  <circle class="circle" cx="10" cy="10" r="10">
</svg>
```

Styling our graphical structures can now be achieved using CSS by including a stylesheet with the following content:

```
.circle {
  fill: black;
}
```

This is fantastic, as we can now not only reuse some graphical structures, but also apply different visual appearance parameters using CSS, similar to those enlightening moments when we managed to reuse some semantic HTML by only changing some CSS.

Let's look at the most important CSS properties we can use to style SVG shapes:

- `fill`: While working with solid SVG shapes, there's always a shape fill and stroke option available; the `fill` property specifies the color of the shape fill.
- `stroke`: This property specifies the color of the SVG shape's outline.
- `stroke-width`: This property specifies the width of the SVG shape's outline on solid shapes. For nonsolid shapes, such as lines, this can be thought of as line width.
- `stroke-dasharray`: This specifies a dash pattern for strokes. Dash patterns are space-separated values that define a pattern.
- `stroke-dashoffset`: This specifies an offset for the dash pattern, which is specified with the `stroke-dasharray` property.
- `stroke-linecap`: This property defines how line caps should be rendered. They can be rendered as square, butt, or rounded caps.
- `stroke-linejoin`: This property specifies how lines are joined together within a path.
- `shape-rendering`: Using this property, you can override the shape-rendering algorithm that, as the name suggests, is used to render shapes. This is particularly useful if you need crispy edges on your shapes.

For a complete reference of the available appearance-relevant SVG attributes, visit the Mozilla Developer website at https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute.

I hope this brief introduction gave you a better feeling about SVG and the great power it comes with. In this chapter, we're going to use some of that power to create nice, interactive graphical components. If you would like to learn more about SVG, I strongly recommend that you go through the great articles by *Sara Soueidan*.

Building SVG components

When building Angular components with SVG templates, there are a couple of things we need to be aware of. The first and most obvious one, is XML namespaces. Modern browsers are very intelligent when parsing HTML. Besides being probably the most fault-tolerant parser in the history of computer science, DOM parsers are very smart in recognizing markup and then deciding how to treat it. They will automatically decide the correct namespaces for us, based on element names, so we don't need to deal with them when writing HTML.

If you've messed around with the DOM API a bit, you would've probably recognized that there are two methods for creating new elements. In the document object, for example, there's a `createElement` function, but there's also `createElementNS` that accepts an additional namespace URI parameter. Also, every created element has a `namespaceURI` property that tells you the namespace of the specific element. This is important since HTML5 is a standard that consists of at least three namespaces:

- **HTML**: This is the standard HTML namespace with the http://www.w3.org/1999/xhtml URI.
- **SVG**: This embraces all SVG elements and attributes and uses the http://www.w3.org/2000/svg URI. You can sometimes see this namespace URI in an `xmlns` attribute of the `svg` elements. In fact, this is not really required, as the browser is smart enough to decide on the correct namespace itself.
- **MathML**: This is an XML-based format to describe mathematical formulas and is supported in most modern browsers. It uses the http://www.w3.org/1998/Math/MathML namespace URI.

We can mix all these elements from different standards and namespaces within a single document, and our browser will figure out the correct namespace itself when it creates elements within the DOM.

Tip

If you want more information on namespaces, I recommend that you go through the *Namespaces Crash Course* article on the Mozilla Developer Network at https://developer.mozilla.org/en/docs/Web/SVG/Namespaces_Crash_Course.

As Angular will compile templates for us and render elements into the DOM using the DOM API, it needs to be aware of the namespaces when doing that. Similar to the browser, Angular provides some intelligence for deciding the correct namespace while creating elements. However, there are some situations where you need to help Angular recognize the correct namespace.

To illustrate some of this behavior, let's transform our circle example that we've been working on into an Angular component:

```
@Component({
```

```
  selector: 'awesome-circle',
  template: `
    <svg [attr.width]="size" [attr.height]="size">
      <circle [attr.cx]="size/2" [attr.cy]="size/2"
              [attr.r]="size/2" fill="black" />
    </svg>
  `
})
export class AwesomeCircle {
  @Input() size;
}
```

We've wrapped our circle SVG graphics into a simple Angular component. The `size` input parameter determines the actual width and height of the circle by controlling the SVG's `width` and `height` attributes and the circle's `cx`, `cy`, and `r` attributes.

To use our `Circle` component, simply use the following template within another component:

```
<awesome-circle [size]="20"></awesome-circle>
```

Note

It's important to note that we need to use attribute bindings on SVG elements, and we can't set DOM element properties directly. This is due to the nature of SVG elements that have special property types—for example, `SVGAnimatedLength`—that can be animated with **Synchronized Multimedia Integration Language** (**SMIL**). Instead of interfering with these rather complex element properties, we can simply use attribute bindings to set the attribute values of the DOM element.

Let's go back to our namespace discussion. Angular would know that it needs to use the SVG namespace to create the elements within this template. It will function in this way simply because we're using the `svg` element as a root element within our component, and it could switch the namespace within the template parser for any child elements automatically.

However, there are certain situations where we need to help Angular determine the correct namespace for the elements we'd like to create. This strikes us if we're creating nested SVG components that don't contain a root `svg` element:

```
@Component({
  selector: '[awesomeCircle]',
  template: `
      <svg:circle [attr.cx]="size/2" [attr.cy]="size/2"
                  [attr.r]="size/2" fill="black" />
  '
})
export class AwesomeCircle {
  @Input('awesomeCircle') size;
}

@Component({
  selector: 'app'
```

```
  template: `
   <svg width="20" height="20">
    <g [awesomeCircle]="20"></g>
   </svg>
  `,
  directives: [AwesomeCircle]
})
export class App {}
```

In this example, we're nesting SVG components, and our `AwesomeCircle` component does not have an `svg` root element to tell Angular to switch the namespace. This is why we've created the `svg` element within our `App` component and then included the `AwesomeCircle` component in an SVG group.

We need to explicitly tell Angular to switch to the SVG namespace within our `Circle` component, and we can do this by including the namespace name as a prefix separated by a colon, as you can see in the highlighted section of the preceding code excerpt.

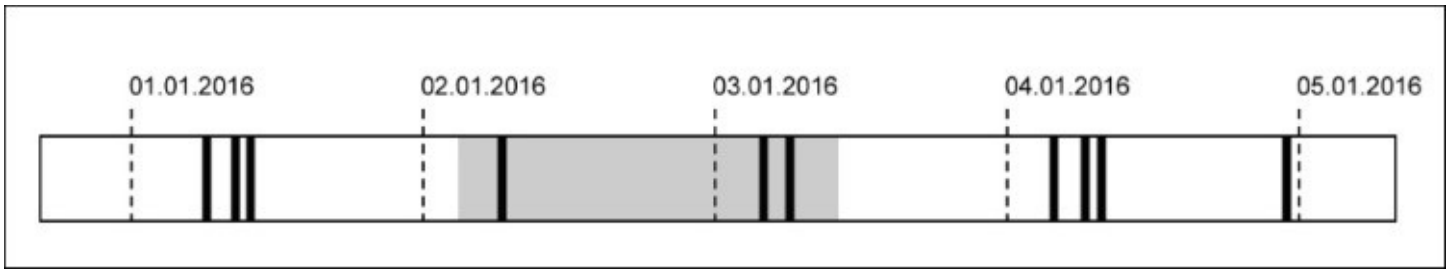
If you have multiple elements that need to be created within the SVG namespace explicitly, you can rely on the fact that Angular does apply the namespace for child elements too and does group all your elements with an SVG group element. So, you only need to prefix the group element `<svg:g> ... </svg:g>`, but none of the contained SVG elements.

This is enough to know about Angular internals when dealing with SVG. Let's move on and create some real components!

Building an interactive activity slider component

In the previous topics, we've covered the basics of working with SVG and dealing with SVG in Angular components. Now it's time to apply our knowledge to the task management application and create some components using SVG.

The first component we'll be creating in this context is an interactive slider that allows the user to select the time range of activities that he or she is interested to check out. Displaying a simple HTML5 range input could be a solution, but since we've gained some SVG superpower, we can do better! We'll use SVG to render our own slider that will show existing activities as ticks on the slider. Let's look at a mock-up of the slider component that we're going to create:



A mockup of the activity slider component

Our slider component will actually serve two purposes. It should be a user control and should provide a way to select a time range for filtering activities. However, it should also provide an overview of all the activities so that a user can filter the range more intuitively. By drawing vertical bars that represent activities, we can already give the user a feeling of the range he or she is interested in.

First of all, we'll create a new file for our `ActivitySlider` component called `activity-slider.js` within the `activities/activity-slider` path and define our component class:

```
import styles from './activity-slider.css!text';

@Component({
  selector: 'ngc-activity-slider',
  host: {
    class: 'activity-slider'
  },
  styles: [styles],
  encapsulation: ViewEncapsulation.Native,
  …
})
export class ActivitySlider {
```

```
  // The input expects a list of activities
  @Input() activities;
  // If the selection of date range changes within our slider
  // component, we'll emit a change event
  @Output() selectionChange = new EventEmitter();

  constructor(@Inject(ElementRef) elementRef) {
    // We'll use the host element for measurement when drawing
    // the SVG
    this.sliderElement = elementRef.nativeElement;
    // The padding on each side of the slider
    this.padding = 20;
  }

  ngAfterViewInit() {
    // We'll need a reference to the overlay rectangle so we can
    // update its position and width
    this.selectionOverlay = this.sliderElement
     .shadowRoot.querySelector('.selection-overlay');
  }
}
```

The first thing we should mention, and which differs from all the other components we've written so far, is that we're using `ViewEncapsulation.Native` for this component. As we learned from the *Creating our application component* section in Chapter 2, *Ready, Set, Go!*, when we use `ViewEncapsulation.Native` for our component encapsulation, Angular actually uses Shadow DOM to create the component. We briefly looked at this in the *Shadow DOM* section in Chapter 1, *Component-Based User Interfaces* as well.

Using Shadow DOM for our component will give us this advantage: our component will be fully encapsulated from the CSS side of things. This not only means that none of the global CSSes will leak into our component, but it also means that we'll need to create local styles in order to style our component.

So far, we've used a CSS naming convention called BEM that provides us with some necessary prefixes to avoid name clashes within CSS and establish a clean and simple CSS specificity. However, when using Shadow DOM, we can forego prefixes to avoid name clashes, since we're only applying styles locally within the component.

Because we're using Shadow DOM for this component, we need to have a way to define local styles. Angular provides us with an option to pass styles into the component using the `styles` property of the component annotation.

Tip

Chrome supports Shadow DOM natively since version 35. Within Firefox, Shadow DOM can be enabled by visiting the `about:config` page and turning on the `dom.webcomponents.enabled` flag. IE, Edge, and Safari don't support this standard at all; however, we can set things up in a way that they could deal with Shadow DOM, by including a polyfill named `webcomponents.js`. You can find more information on this polyfill at

https://github.com/webcomponents/webcomponentsjs.

Using the text plugin of SystemJS, we can import a stylesheet containing only the local styles of our component and then pass them to the `styles` property. By appending a `!text` postfix to the import of our CSS file, we tell SystemJS to load our CSS file as raw text. Note that the `styles` property is expecting an array, which is why we wrap our imported styles into an array literal.

If you take a look at the stylesheet for the `ActivitySlider` component, you can immediately see that we're no longer prefixing the classes with the component name:

```
.slide {
  fill:#f9f9f9;
}

.activity {
  fill:#3699cb;
}

.time {
  fill:#bbb;
  font-size:14px;
}

.tick {
  stroke:#bbb;
  stroke-width:2px;
  stroke-dasharray:3px;
}

.selection-overlay {
  fill:#d9d9d9;
}
```

Usually, such short class names would probably lead to name clashes within our project, but since the styles will be local to the Shadow DOM of our component, we don't need to worry about name clashes any more.

As an input parameter, we define the list of activities that will be used not only to determine the available range in the slider, but also to render activities on the background of the slider.

Once a selection is made by the user, our component will use the `selectionChange` event emitter to notify the outside world about the change.

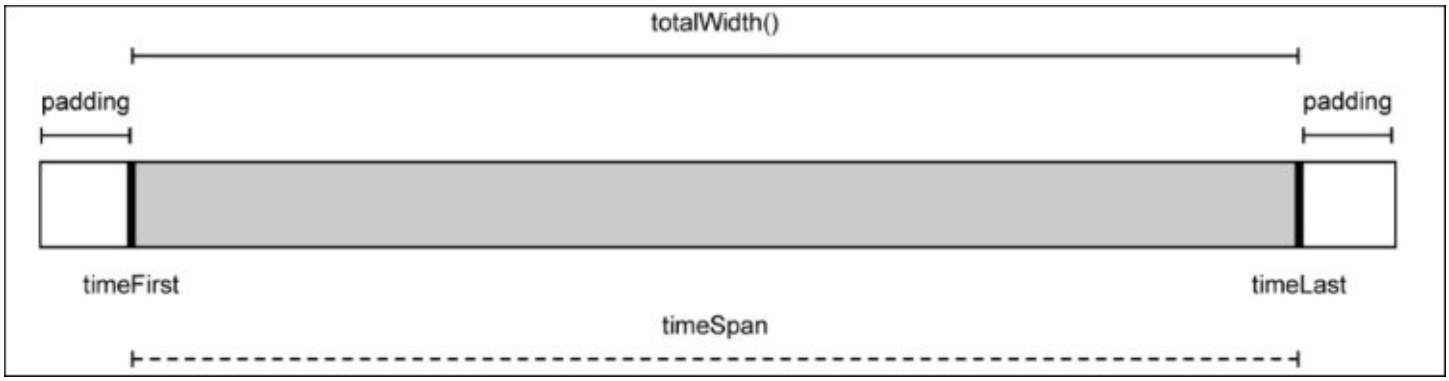
Within the constructor, we're setting aside the component DOM element for some measurement we need to make in order to draw later on:

```
this.sliderElement = elementRef.nativeElement;
```

By injecting the `ElementRef` instance to the constructor, we can easily access the native DOM element of our component.

Projection of time

Our slider component needs to be able to project timestamps into the coordinate system of SVG. Also, when a user clicks on the timeline to select a range, we'll need to be able to project coordinates back into timestamps. For this purpose, we need to create two projection functions within our component that will use a few helper functions and states to calculate the values, from coordinates to time and vice-verse:



Visualization of important variables and functions for our calculations

While we will use percentage to position our SVG elements on the slider component, the padding on the sides will need to be specified in pixels. The `totalWidth` function will return the total width of the area in pixels; this is where we'll draw the activity indicators. The `timeFirst`, `timeLast`, and `timeSpan` variables will also be used by the calculations and are specified in milliseconds.

Let's add some code to our slider to deal with the projection of our activities on the slider in the `activity-slider.js` file:

```
// Getting the total available width of the slider
totalWidth() {
  return this.sliderElement.clientWidth - this.padding * 2;
}

// Projects a time stamp into percentage for positioning
projectTime(time) {
  let position = this.padding +
    (time - this.timeFirst) / this.timeSpan * this.totalWidth();
  return position / this.sliderElement.clientWidth * 100;
}

// Projects a pixel value back to a time value. This is required
// for calculating time stamps from user selection.
projectLength(length) {
  return this.timeFirst + (length - this.padding) / this.totalWidth() *
this.timeSpan;
}
```

Since we have put aside the reference to the root element as the `sliderElement` member variable, we can use its `clientWidth` property to get the full width of the component and subtract the padding. This will give us the full width of the area where we'd like to draw activity indicators, in pixels.

In the `projectTime` function, we will first transform the timestamp into a position by a simple rule of three. Because we have access to the timestamp of the first activity as well as the total time span, this will be quite a simple task. Once we do this, we can convert our position value, which is of unit pixels, into percentage, by dividing it by the total component width and then multiplying it by 100.

To project a pixel value back to a timestamp, we can do more or less the reverse of `projectTime`, except that we're not dealing with percentage here but assuming the length parameter of the `projectLength` function is in pixel unit.

We've used some member variables—`timeFirst`, `timeLast`, and `timeSpan`—within our projection code, but how do we set these member variables? Since we have an `activities` component input, which is expected to be a list of relevant activities, we can observe the input for changes and set the values based on the input. To observe component input for changes, we can use the `ngOnChanges` life cycle hook:

```
ngOnChanges(changes) {
  // If the activities input changes we need to re-calculate and
  // re-draw
  if (changes.activities && changes.activities.currentValue) {
    const activities = changes.activities.currentValue;
    // For later calculations we set aside the times of the
    // first and the last activity
    if (activities.length === 1) {
      // If we only have one activity we use the same time for
      // first and last
      this.timeFirst = this.timeLast = activities[0].time;
    } else if (activities.length > 1) {
      // Take first and last time
      this.timeFirst = activities[activities.length - 1].time;
      this.timeLast = activities[0].time;
    } else {
      // No activities yet, so we use the current time for both
      // first and last
      this.timeFirst = this.timeLast = new Date().getTime();
    }

    // The time span is the time from the first activity to the
    // last activity. We need to limit to lower 1 for not messing
    // up later calculations.
    this.timeSpan = Math.max(1, this.timeLast - this.timeFirst);
  }
}
```

First, we need to check whether the changes include changes to the `activities` input and that the current value of the input is valid. After checking for the input value, we can determine

our member variables, namely `timeFirst`, `timeLast`, and `timeSpan`. We limit the `timeSpan` variable to `1` at least, as our projection calculations would be messed up otherwise.

The preceding code will ensure that we will always recalculate our member variables when the `activities` input changes and that we'd be using the most recent data-rendering activities.

Rendering activity indicators

We've already implemented the basics of the component and laid the groundwork for drawing time information into the coordinate system of our component. It's time to use our projection functions and draw our activities as indicators on the slider using SVG.

First, let's take a look at the required template that we are going to create in a file called `activity-slider.html` within our `activity-slider` directory:

```
<svg width="100%" height="70px">
  …
  <rect x="0" y="30" width="100%" height="40"
        class="slide"></rect>
  <rect *ngFor="let activity of activities"
        [attr.x]="projectTime(activity.time) + '%'"
        height="40" width="2px" y="30" class="activity"></rect>
</svg>
```

Since we need to create an indicator for every activity within our activities list, we can simply use the `NgFor` directive to repeat the rectangle that represents our activity indicator.

As we know from building our `ActivityService` class in a previous topic, activities always contain a `time` field with the timestamp of the activity. Within our component, we have already created a projection function that converts time into percentage, relative to our component width. We can simply use the `projectTime` function within our attribute binding for the `x` attribute of the `rect` element to position our activity indicators at the correct positions.

By using only an SVG template and our backing function to project time, we have created a nice little chart that displays activity indicators on a timeline.

You can imagine that if we have a lot of activities, our slider will actually look pretty stuffed, and it will be hard to get a feeling for when those activities may have occurred. We need to have some sort of a grid that will help us associate the chart with a timeline.

As already shown in the mock-up of our slider component, we're now going to introduce some ticks on the slider background that will divide the slider into sections. We'll also label each tick with a calendar time. This will give our users a rough sense for time when looking at the activity indicators on the slider.

Let's look at the code changes within our `ActivitySlider` class that will enable the rendering of our ticks:

```
ngOnChanges(changes) {
  // If the activities input changes we need to re-calculate and
  // re-draw
  if (changes.activities && changes.activities.currentValue) {
    ...
    // Re-calculate the ticks that we display on top of the slider
    this.computeTicks();
```

```
  ...
}

// This function computes 5 ticks with their time and position on
// the slider
computeTicks() {
  const count = 5;
  const timeSpanTick = this.timeSpan / count;
  this.ticks = Array.from({length: count}).map(
    (element, index) => {
      return this.timeFirst + timeSpanTick * index;
    });
}
...
```

First of all, we need to create a function that computes some ticks for us that we can place onto the timeline. For this purpose, we need to create the `computeTicks` method that will divide the whole timeline into five equal segments and generate timestamps that represent the position in time for individual ticks. We store these ticks in a new `ticks` member variable. With the help of these timestamps, we can render the ticks within our view easily.

Tip

We use the `Array.from` ES6 function to create a new array with the desired length, and use the functional array extra function `map` to generate tick model objects from this array. Using `Array.from` is a nice trick to create an initial array of a given length that can be used to establish functional style.

Let's look at the template of our component and how we can use our array of timestamps to render ticks on our slider component. We're going to modify our component template in `activity-slider.html`:

```
...
<g *ngFor="let tick of ticks">
  <text [attr.x]="projectTime(tick) + '%'" y="14" class="time">
    {{tick | calendarTime}}</text>
  <line [attr.x1]="projectTime(tick) + '%'"
        [attr.x2]="projectTime(tick) + '%'"
        y1="30" y2="70" class="tick"></line>
</g>
...
```

To render our ticks, we've used an SVG group element to place our `NgFor` directive that repeats the tick timestamps we've stored in the `ticks` member variable.

For each tick, we need to place a label as well as a line that spans over the slider background. We can use the SVG text element to render our label with the timestamp on top of the slider. Within the attribute binding for the `x` attribute of our `text` element, we've used our `projectTime` projection function to receive the projected percentage value from our timestamp. The `y` coordinate of our `text` element is fixed at a position where the labels will

just sit on top of our slider.

SVG lines consist of four coordinates: `x1`, `x2`, `y1`, and `y2`. Together they define two coordinate points where a line will be drawn from one point to the other.

Now we are getting closer to the final slider that we specified in the mock-up at the beginning of this topic. The last missing piece of the puzzle is to make our slider interactive so a user can select a range of activities.

Bringing it to life

So far, we've covered the rendering of the slider background as well as the rendering of the activity indicators. We've also generated ticks and displayed them with a grid line and a label to display the calendar time of each tick.

Well, that does not really make a slider, does it? Of course, we also need to handle user input and make the slider interactive so users can select a time range they want to display the activities for.

To do this, add the following changes to the component class:

```
ngOnChanges(changes) {
  // If the activities input changes we need to re-calculate and
  // re-draw
  if (changes.activities && changes.activities.currentValue) {
    ...
    // Setting the selection to the full range
    this.selection = {
      start: this.timeFirst,
      end: this.timeLast
    };
    // Selection changed so we need to emit event
    this.selectionChange.next(this.selection);
  }
}
```

When we detect a change in the `activities` input property within the `ngOnChanges` life cycle hook, we initialize a model for the user selection in our slider component. It consists of a `start` and `end` property, both containing timestamps that represent the selected range on our activity slider.

Once we've set our initial selection, we need to use the `selectionChange` output property to emit an event. This way, we can let our parent component know that the selection within the slider has changed.

To display the selected range, we use an overlay rectangle within our template that will be placed above the slider background. If you look at the mock-up image of the slider again, you'll notice that this overlay is painted in gray:

```
  <rect *ngIf="selection"
        [attr.x]="projectTime(selection.start) + '%'"
        [attr.width]="projectTime(selection.end) -
projectTime(selection.start) + '%'"
        y="30" height="40" class="selection-overlay"></rect>
```

This rectangle will be placed just above our slider background and will use our projection function to calculate the `x` and `width` attributes. As we need to wait for change detection to initialize our selection within the `ngOnChanges` life cycle hook, we'll just check for a valid

selection object by making use of the `NgIf` directive.

Now we need to start tackling user input in our `ActivitySlider` component. The mechanics for storing the state and rendering our selection is already in place, so we can implement the required host listeners to handle user input:

```
...
// If the component receives a mousedown event, we need to start a
// new selection
@HostListener('mousedown', ['$event'])
onMouseDown(event) {
  // Starting a new selection by setting selection start and end
  // to the projected time of the clicked position.
  this.selection.start = this.selection.end =
    this.projectLength(event.offsetX);
  // Selection changed so we need to emit event an
  this.selectionChange.next(this.selection);
  // Setting a flag so we know that the user is currently moving
  // the selection
  this.modifySelection = true;
}

// We also need to track mouse moves within our slider component
@HostListener('mousemove', ['$event'])
onMouseMove(event) {
  // We should only modify the selection if the component is in
  // the correct mode
  if (this.modifySelection) {
    // Update the selection end with the projected time from the
    // mouse coordinates
    this.selection.end = Math.max(this.selection.start,
      this.projectLength(event.offsetX));
    // Selection changed so we need to emit event an
    this.selectionChange.next(this.selection);
    // To prevent side effects, we should stop propagation and
    // prevent browser default
    event.stopPropagation();
    event.preventDefault();
  }
}

// If the user is releasing the mouse button, we should stop the
// modify selection mode
@HostListener('mouseup')
onMouseUp() {
  this.modifySelection = false;
}

// If the user is leaving the component with the mouse, we should
// stop the modify selection mode
@HostListener('mouseleave')
onMouseLeave() {
  this.modifySelection = false;
}
...
```

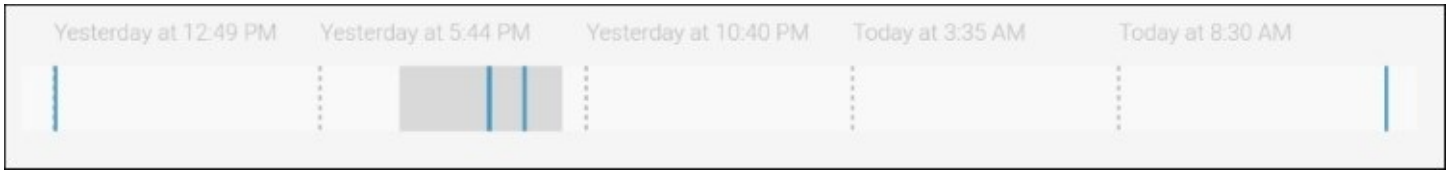
In the preceding code excerpt, we handled a total of four events on the slider host element:

- `onMouseDown`: We set our selection model's `start` and `end` properties with the same value. Since we're using timestamps for these properties, we projected the mouse position into the timespace first. The mouse position comes in pixels relative to the slider component's origin. Since we know the slider's width and the total time duration displayed, we can convert this into timestamps easily. We're using the `projectLength` method for this purpose. By passing a second argument to the `@HostListener` decorator, we specified that we'd like to pass the DOM event to our `onMouseDown` method. We also set a state flag, `modifySelection`, in our component to indicate that a selection is under progress.
- `onMouseMove`: If the component is in selection mode (the `modifySelection` flag is `true`), you can adjust the end property of the `selection` object. Here, we also made sure that we ruled out the possibility of creating a negative selection by using `Math.max` and limiting the end of the selection to not be smaller than the start.
- `onMouseUp`: When the user releases the mouse button, the component exits the selection mode. This can be done by setting the `modifySelection` flag to `false`.
- `onMouseLeave`: This is the same as the `onMouseUp` event; the difference is that here the component will just exit the selection mode.

Using the `@HostListener` decorator, we were able to handle all of the necessary user input to complete our component with the interactive elements that were still missing.

Recap

In this topic, we learned how to use SVG in order to create graphical and interactive components with Angular. By creating attribute bindings on our SVG elements and controlling the instantiation of graphical elements using the `NgFor` and `NgIf` directives, we built a custom slider component that provides a nice overview of our activities. At the same time, we also learned how to handle user input using the `@HostListener` decorator in order to make our component interactive:



A screenshot of the finished activity slider component

To sum things up, we learned about the following concepts:

- Encapsulating component views using `ViewEncapsulation.Native` and importing local styles
- Covering some basic projections of timestamps onto screen coordinates to be used with SVG elements
- Handling user input and creating a custom selection mechanism using the `@HostListener` decorator

Building the activity timeline

So far, we've built a service to log activities and a slider component to select a time range and provide an overview using activity indicators. Since we needed to perform a lot of drawing tasks within the slider component, SVG was a perfect fit for this use case. To complete our `Activities` component tree, we still need to render the activities that were selected using the `ActivitySlider` component.

Let's continue to work on our activities component tree. Well create a new component that will be responsible for rendering an individual activity within an activity timeline. Let's start with the template of the `Activity` component, which we will create in a new `activities/activity/activity.html` file:

```
<img [attr.src]="activity.user.pictureDataUri"
     [attr.alt]="activity.user.name"
     class="activity__user-image">
<div [class.activity__info--align-right]="isAlignedRight()"
     class="activity__info">
  <h3 class="activity__title">{{activity.title}}</h3>
  <p class="activity__author">
    by {{activity.user.name}} {{activity.time | fromNow}}
  </p>
  <p>{{activity.message}}</p>
</div>
```

Each activity will consist of a user image as well as an information box that will contain the activity title, message, and authoring details.

Our activity will use an input to determine its alignment. This allows us to align the activity from outside the component. The `isAlignedRight` method helps us set an additional CSS class, `activity__info--align-right`, on the activity information box.

We also need to create a component class for our `Activity` component, which we will create under a new `activities/activity/activity.js` file:

```
import {FromNowPipe} from '../../pipes/from-now';

@Component({
  selector: 'ngc-activity',
  …
  // We are using the FromNow pipe to display relative times
  // within our template
  pipes: [FromNowPipe]
})
export class Activity {
  @Input() activity;
  // Input that should be a string 'left' or 'right' and will
  // determine the activity alignment using CSS
  @Input() alignment;
  @Input() @HostBinding('class.activity--start-mark') startMark;
```

```
  @Input() @HostBinding('class.activity--end-mark') endMark;

  // Function with that will tell us if the activity should be
  // aligned to the right. It's used for setting a modifier class
  // on the info element.
  isAlignedRight() {
    return this.alignment === 'right';
  }
}
```

Our `Activity` component expects four inputs:

- `activity`: This property takes the data model of the activity that needs to be rendered with the component. This is the activity that we created using `ActivityService`.
- `alignment`: This input property should be set to a string containing the word `left` or `right`. We used this to determine whether we needed to add an additional CSS class to our template in order to align the activity information box to the right.
- `startMark`: This input property acts as an input and a host binding at the same time. If this input is set to `true`, the activity will get an additional CSS class, `activity--start-mark`, which will cause a small mark on top of the timeline to indicate the timeline termination.
- `endMark`: In the same way as `startMark`, this input uses a host binding to set an additional CSS class, `activity--end-mark`, which will cause a small mark on the bottom of the timeline to indicate the timeline termination.

The `isAlignedRight` method is used within the template to determine whether we need to add an additional CSS class to the information box in order to align it to the right.

We formatted the timestamp of the activity using the `FromNow` pipe, which we created in Chapter 4, *No Comments, Please!*. In order to use the pipe in the template, we need to import it and add it to the `pipes` property of our component annotation.

We now have almost all the components to display our activities. But still, there's something missing, which is the glue to combine `ActivitySlider` with our `Activity` components and also make our component subtree navigable. For this, we'll create a new component called `Activities`. Let's create an `activities/activities.js` file to write our component class:

```
@Component({
  selector: 'ngc-activities',
  ...
  directives: [ActivitySlider, Activity]
})
export class Activities {
  @Input() activitySubject;

  constructor(@Inject(ActivityService) activityService) {
    this.activityService = activityService;
  }

  ngOnChanges(changes) {
    if (changes.activitySubject) {
      // If we have a subscription to the activities service
```

```
      // already we need to unsubscribe first
      if (this.activitiesChangeSubscription) {
        this.activitiesChangeSubscription.unsubscribe();
      }

      // When the project data is updated we need to filter for
      // activities again
      this.activitiesChangeSubscription =
        this.activityService.change.subscribe((activities) => {
        // Filter for all activities that have the project ID as subject
        this.activities = activities
          .filter((activity) => activity.subject ===
this.activitySubject.document.data._id);
        this.onSelectionChange();
      });
    }
  }
```

First of all, we need to know which activities we want to display within our component. For this, we need to provide a component input, namely `activitySubject`. Once this is done, we can pass an activity subject from the parent component and use it to filter activities we're interested in.

Since we've used activity subjects to log activities as well, we can use the same subjects to display activities. In the `ngOnChanges` life cycle hook, we set up a subscription on the `ActivityService` instance to react to newly created activities. Because the activity service will notify us with an updated list of activities, we can simply use the `Array.prototype.filter` function to filter only relevant items. We've made use of the `activitySubject` input to obtain the ID from the subject.

Next, we need to create a method to apply a date range filter to our activities. The `onSelectionChange` method will be called from our `activities` template, where we created a binding to our `ActivitySlider` component:

```
  // If the selection within the activity slider changes, we need
  // to filter out activities again
  onSelectionChange(selection = this.selection) {
    this.selection = selection;
    // Store filtered activities that fall into the date range
    // selection specified by the slider
    this.selectedActivities = this.selection ? this.activities.filter(
      (activity) => activity.time >= this.selection.start
                 && activity.time <= this.selection.end
    );
  }
```

Whenever the time range is updated by the user within the slider, we'll override the `selectedActivities` member variable with a new filtered version of the activities, which we'd obtain from `ActivityService`. The filter will narrow down the activities by comparing the activity time against the selection range from the slider component.

Now we will set up some helper functions to be used within our template:

```
  // Get an alignment string based on the index. Activities with
  // even index get aligned left while odds get aligned right.
  getAlignment(index) {
    return index % 2 === 0 ? 'left' : 'right';
  }

  // Function to determine if an activity index is first
  isFirst(index) {
    return index === 0;
  }

  // Function to determine if an activity index is last
  isLast(index) {
    return index === this.selectedActivities.length - 1;
  }
```

The three methods, namely `getAlignment`, `isFirst`, and `isLast`, are used within the template as input for the `Activity` component. If you take a look at the code of `ActivityComponent` again, you will see that we need to provide some input in order to set some CSS classes for appearance. The three methods we created here will be used for this purpose:

```
  // If the component gets destroyed, we need to unsubscribe from
  // the activities change observer
  ngOnDestroy() {
    this.activitiesChangeSubscription.unsubscribe();
  }
}
```

Finally, we added an `OnDestroy` life cycle hook that will unsubscribe us from the activity's change observable.

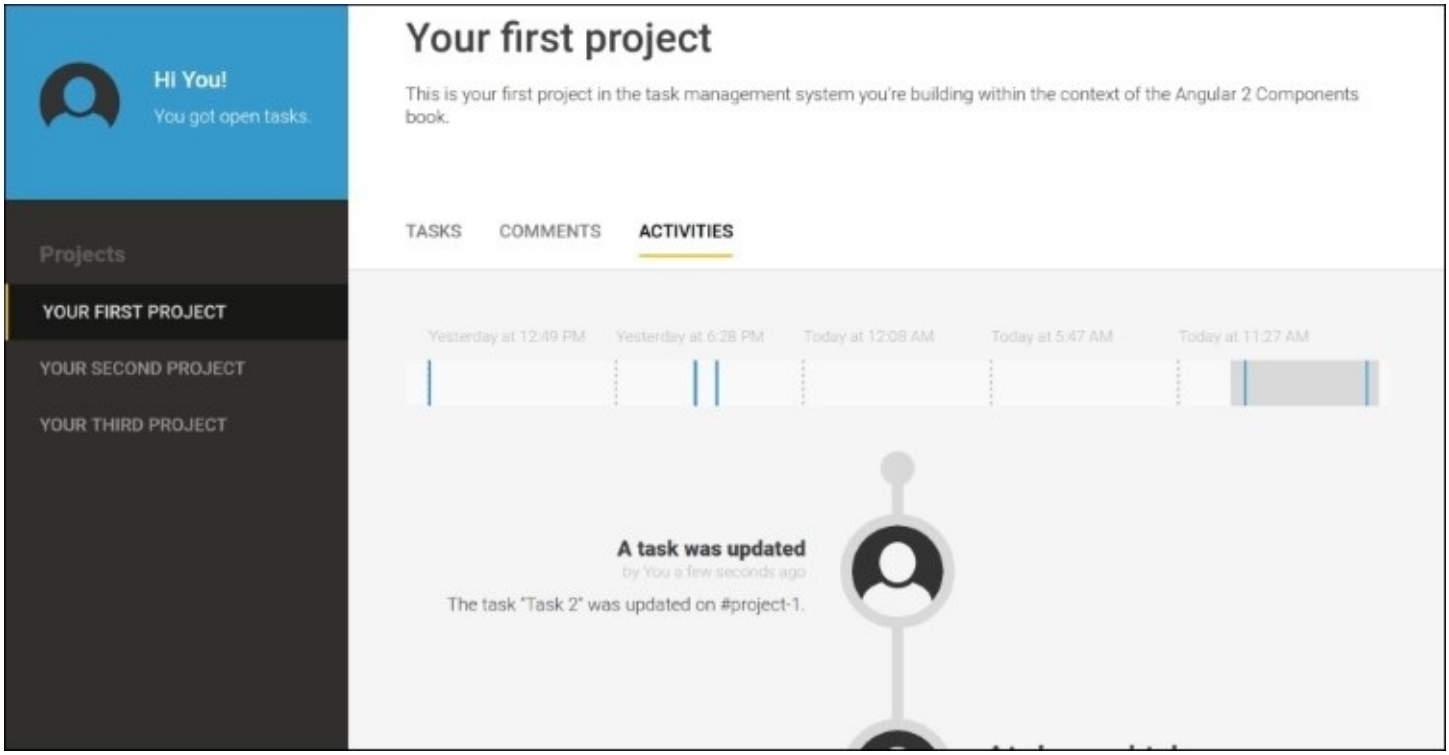
The template for this component is rather simple. The only thing we need to do is render the `ActivitySlider` component, as well as iterate over the selected activities and wire in the `Activity` component for each iteration:

```
<ngc-activity-slider [activities]="activities"
                     (selectionChange)="onSelectionChange($event)">
</ngc-activity-slider>
<div class="activities__l-container">
  <ngc-activity
   *ngFor="let activity of selectedActivities, let index = index"
   [activity]="activity"
   [alignment]="getAlignment(index)"
   [startMark]="isLast(index)"
   [endMark]="isFirst(index)">
  </ngc-activity>
</div>
```

There's not much we need to explain here. We've bound `activities` and our `onSelectionChange` method to the slider component and iterated over all the selected activities to render our `Activity` components. We've created a local view variable, `index`, which we

will use for the appearance input of the `Activity` component.

That's it for our activities page! We've created three components that are composed together and display an activity stream, which provides a slider to filter activities for dates:



A screenshot of the finished activities view

Summary

In this chapter, we created an interactive slider component using SVG. While doing this, we learned about some SVG basics and the power of SVG within the DOM. Using Angular, we were able to make SVG composable, which it isn't by nature. We learned about namespaces, how Angular handles them, and how we can tell Angular that we'd like to use namespaces explicitly.

Besides using SVG for our slider component, we also learned how to use Shadow DOM to create native view encapsulation. As a result of this, we were able to use local styles for our component. We don't need to worry about CSS name clashes, specificity, and global CSS side effects any more when using local styles.

The whole code for this chapter can be found in the ZIP file of the book resources that you can download from the Packt Publishing website.

In the next chapter, we're going to enhance what we've built throughout the chapters so far. We will create some components to enrich the user experience within our application.

Chapter 7. Components for User Experience

User experience is a core concern for developers building today's applications. We are no longer living in a world where users are contented with an application that just works. The expectations are much higher. Now, an application needs to be highly usable and should provide an efficient workflow; users also expect it to bring them pleasure while performing tasks.

In this chapter, we're going to look at building some components that will increase the overall usability of our task management system. These features will enrich the current functionality and provide more efficient workflows.

We will develop the following three technical features and embed them into our current application, wherever applicable:

- **Tag management**: We'll enable the use of tags within generated content, such as comments, activities, and other areas where they can be of any use. Tags will help users build links between content and navigation shortcuts.
- **Drag and drop**: We'll build generic components that will make use of drag and drop features a breeze. By enabling drag and drop features, we'll allow users to fulfill certain tasks with much higher efficiency.
- **Infinite scrolling**: We'll build a component that will reveal the content of lists while scrolling. This feature is not going to directly increase the workflow performance, but it will help us increase the overall application performance. It will also narrow down the user's context by only showing relevant information.

We'll cover the following topics in this chapter:

- Creating a tag management system to enter and display tags
- Creating a stateful pipe to render tags using a service
- Using the `sanitize-html` module to sanitize potentially unsafe content
- Creating a component to autocomplete tags during user input
- Going through the basics of the HTML5 drag and drop API
- Creating directives for draggable elements and drop targets
- Using `dataTransfer` objects and a custom attribute to enable selective drop targets
- Creating a custom `ForOf` repeater using the asterisk template syntax to enable infinite scrolling
- Implementing custom change detection using the `DoCheck` lifecycle hook, and using `IterableDiffer` to apply DOM changes
- Performing dynamic view instantiation using `ViewContainer`

Tag management

The classical form of tagging enables you to associate a taxonomy with elements within a system and helps you organize your project. It allows you to have a many-to-many association that can be quickly managed, and you can use it later to filter relevant information.

In our task management system, we're going to use a slightly different version of tags. Our goal is to provide a way to have semantic shortcuts within the application. With the help of tags, a user should be able to cross-reference information between different parts of the data, providing a summary of the referenced entity as well as a navigation shortcut for the entity.

For example, we can include a `project` tag within a user comment. A user can enter the tag by simply typing in the project ID. When a comment is displayed, we see the title of the project and the number of open tasks within the project. But when we click on the tag, we directly reach the project detail page where the task is located.

In this section, we'll develop the required elements to provide a way to use `project` tags that will enable the user to cross-reference other projects within comments. We'll also use tag management in our activities, which we created in the previous chapter.

Tag data entity

Let's start with the tag entity that shows how we can represent tags within our system. We'll create a new `Tag` class in a file under `tags/tag.js`:

```
// Class that represents a tag
export class Tag {
  constructor(textTag, title, link, type) {
    // The textTag property is the text representation of the tag
    this.textTag = textTag;
    this.title = title;
    this.link = link;
    this.type = type;
  }
}
```

This class represents tags; whenever we store tag information, we'll use this entity as a data vehicle. Let's look at the individual fields and elaborate on their use:

- `textTag`: This is the text representation of a tag. All our tags need to be identified uniquely using this text representation. We can define the text representation of tags as follows:
  - Text tags always start with a hash symbol (`#`)
  - Text tags only contain word characters or the minus symbol (`-`)
  - All the innards of a tag, defined by other properties (`title`, `link`, and `type`), can be extrapolated from the `textTag` property. It can therefore be considered an ID.
- `title`: This is a comparatively longer text representation of a tag. It should contain as much detail about the subject as possible. In the case of project tags, this could mean the project title, open tags count, assignee, and other important information. Since this is the field that will be rendered if a tag is parsed, it'll be beneficial if the content stays relatively condensed.
- `link`: A valid URL, which will be used when the tag is rendered. This URL will make links clickable and enable the shortcut navigation. In the case of the projects tags we're going to create, this will be a URL fragment identifier that will link to the given project page.
- `type`: This is used to distinguish between different tags and provide us a way to organize tags at a higher granularity level.

So far, so good. We now have a data vehicle we can easily construct to transfer information about tags.

Generating tags

Our next step is to create a factory that will generate tags for us. All we'd like to pass to the factory is a subject, which can be basically anything. The factory will then determine the type of the subject and execute the necessary logic to generate a tag from it. This might sound a bit abstract at first, but let's look at the code of the `generateTag` function we'll create in a module under `tags/generate-tag.js`:

```
import {Tag} from './tag';
import {limitWithEllipsis} from '../utilities/string-utilities';
export const TAG_TYPE_PROJECT = 'project';

// The generateTag function is responsible for generating new tag
// objects depending on the passed subject
export function generateTag(subject) {
  if (subject.type === TAG_TYPE_PROJECT) {
    // If we're dealing with a project here, we generate the
    // according tag object
    const openTaskCount = subject.tasks.filter((task) => !task.done).length;
    return new Tag(
      `#${subject._id}`,
      `${limitWithEllipsis(subject.title, 20)} (${openTaskCount} open tasks)`,
      `#/projects/${subject._id}/tasks`,
      TAG_TYPE_PROJECT
    );
  }
}
```

Let's examine the `generateTag` function and what we're trying to achieve here.

First, we determined the subject type by checking the type attribute of the `subject` object. In the case of project data objects, we know that the type will be set to `"project"`. The following three points succinctly explain what we've done:

1. Since we were sure that we were dealing with a project here, we generated a new tag. In the future, we'll deal with other subject types as well, so this check will be required.
2. We wanted to use an indicator for all the open tasks in the tag title. For this reason, we did a quick filtering of open tasks within the project and stored the length of the filtered array in the `openTaskCount` constant.
3. Now we can instantiate a new `Tag` object using the project ID as `textTag`. For the `title` field, we used a helper function, `limitWithElipsis`, which truncates project titles that are longer than 20 characters. We also appended the open tasks count to the tag title. For the `link` field of the `Tag` instance, we specify a URL that will navigate to the project details view. Finally, we used the `TAG_TYPE_PROJECT` constant to define the tag `type` field.

Creating a tags service

Okay, we're done with setting up all the supporting structures we need; we can now move forward to create a tags service. A tags service will have the following responsibilities:

- **Generating and caching tags**: We won't create tags in our system ad hoc if we only want to render them. The mechanics of a tags service is more like generated cache. Initially, a tags service gathers all the required information to generate all the possible tags within the system. It also reacts to changes and updates the list of tags if required. With this, we'll not only save on some processing needs, but we'll also have a readily available list to search for existing tags. This will be particularly useful if we like to present the available tags to the user so they can choose from them.
- **Rendering tags**: A tags service is also responsible for turning tags into HTML. It uses the `title` and `link` fields of the `Tag` instances to generate their HTML representation.
- **Parsing text content**: The parsing functionality of the tags service is responsible for finding text representations of tags within a string. It then uses the rendering function to render these tags into HTML.

Let's create a module for our tags service in a new file under `tags/tags-service.js`.

First, we need to create two utility functions that will help us process tags and strings containing the textual representations of tags.

The `replaceAll` function is a simple substitute for a missing JavaScript function to replace multiple text occurrences within a string without using regular expressions:

```
// Utility function to replace all text occurrences in a string
function replaceAll(target, search, replacement) {
  return target.split(search).join(replacement);
}
```

The `findTags` function will extract any possible tag from a text string. It does this by applying a regular expression that will find matches for tags in the format discussed at the beginning of the topic. This format assumes that our tags always start with a hash symbol, followed by any word character or dash symbols. This function returns a list of all the possible text tags:

```
// Function to find any tags within a string and return an array
// of discovered tags
function findTags(str) {
  const result = [];
  const regex = /#[\w\/-]+/g;
  let match;
  while (match = regex.exec(str)) {
    result.push(match[0]);
  }
  return result;
}
```

For our tags service, we will now define a new class that will be annotated with `@Injectable`

so we can use it as a provider in our components:

```
@Injectable()
export class TagsService {
...
}
```

Let's look at the constructor of our `TagsService` class:

```
constructor(@Inject(ProjectService) projectService) {
  // If the available tags within the system changes, we will
  // emit this event
  this.change = new ReplaySubject(1);
  // In order to generate project tags, we're making use of the
  // ProjectService
  this.projectService = projectService;
  this.projectService.change.subscribe((projects) => {
    // On project changes we store the new project list and re-
    // initialize our tags
    this.projects = projects;
    this.initializeTags();
  });
}
```

In order to generate and cache project tags, we obviously need `ProjectService`, which provides us with a list of all the projects. Instead of grabbing the list data from `ProjectService` once, we're observing the list for changes. This brings us the advantage that we'll not only get the initial list of projects, but we'll also be made aware of any changes made in the project list.

We subscribed to `ProjectService` using the `change` field. This exposes `ReplaySubject`, which emits the project list. After storing the current project list as a member field, we need to call the `initializeTags` method:

```
// This method is used internally to initialize all available tags
initializeTags() {
  // We're creating tags from all projects using the generateTag
  // function
  this.tags = this.projects.map(generateTag);
  // Since we've updated the list of available tags we need to
  // emit a change event
  this.change.next(this.tags);
}
```

As we only support project tags currently, the only thing we need to consider while generating tags is the projects we have stored in our service. We can simply map the project list we have stored in the `projects` member field using our `generateTag` function. The `Array.prototype.map` function will return a new array that is already a list of generated tasks for the projects.

Rendering tags

Okay, we now have a service that uses a reactive approach to generate tags from the available projects. This is already addressing the first concern of our service. Let's look at its other responsibilities, which are parsing text content for tags and rendering HTML.

Rendering tags is not a big deal since we have already abstracted the data model of tags in a clean way. We need to write a method for rendering tags that will act as a pass-through function if the argument is not a valid `Tag` instance. This way, we can pass unrecognized text representations of tags as strings, and it will just return us the string.

Since tags have URLs that point to a location, we're going to use anchor HTML elements to represent our tags. These elements also have classes that will help us style tags differently than regular content. Let's create another method within the tags service that can be used to render tag objects into HTML:

```
renderTag(tag) {
  if (tag instanceof Tag) {
    return `<a class="tags__tag tags__tag--${tag.type}"
href="${tag.link}">${tag.title}</a>`;
  } else {
    return tag;
  }
}
```

The following method can be used to find a tag by its textual representation. This function will try to find the tag within our generated cache, and if unsuccessful, will return the `textTag` argument. This is also a pass-through mechanism that simplifies the handling when we parse a whole piece of text for tags:

```
// This method will lookup a tag via its text representation or
// return the input argument if not found
parseTag(textTag) {
     return this.tags.find(
    (tag) => tag.textTag === textTag
  ) || textTag;
}
```

Last but not least, let's implement the main method of the service. The parse function scans the whole text for tags and replaces them with their HTML representation:

```
// This method takes some text input and replaces any found and
// valid text representations of tags with the generated HTML
// representation of those tags
parse(value) {
  // First we find all possible tags within the text
  const tags = findTags(value);
  // For each found text tag, we're parsing and rendering them
  // while replacing the text tag with the HTML representation
  // if applicable
  tags.forEach(
```

```
    (tag) => value = replaceAll(value, tag,
                          this.renderTag(this.parseTag(tag))));
  );
  // After all tags have been rendered, we're using a sanitizer
  // to ensure some basic security
  return value;
}
```

First, we need to use the `findTags` utility function; this will return a list of all the text tags that it would find in the string content passed to the `parse` function. Using this text tag list, we can then iterate through the list and successively replace all the text tags in the content with the generated HTML using the `renderTag` method.

Integrating the task service

All the concerns of our task service have now been taken care of, and it is already storing tags for the available projects. We can now go ahead and integrate our service into the application.

Since our tags service turns text with simple hash tags into HTML with links, a pipe would be a perfect helper to integrate the functionality within our components.

Let's create a `tags.js` file in our `pipes` folder and create a new pipe class, namely, `Tags`:

```
import {Pipe, Inject} from '@angular/core';
import {TagsService} from '../tags/tags-service';

@Pipe({
  name: 'tags',
  // Since our pipe is depending on services, we're dealing with a
  // stateful pipe and therefore set the pure flag to false
  pure: false
})
export class TagsPipe {
  constructor(@Inject(TagsService) tagsService) {
    this.tagsService = tagsService;
  }
  // The transform method will be called when the pipe is used within a template
  transform(value) {
    if (typeof value !== 'string') {
      return value;
    }
    // The pipe is using the TagsService to parse the entire text
    return this.tagsService.parse(value);
  }
}
```

We have already created a few pipes so far. However, this pipe is a bit different in that it isn't a pure pipe. Pipes are considered pure if their transform function always returns the same output for a given input. This implies that the transform function should not be dependent on any other external source that can influence the outcome of the transform, and the only dependencies are the input values. This is not true for our `Tags` pipe though. It depends on `TagsService` to transform the input, and new tags can be stored in the tags service at any time. Successive transformations can successfully render tags that were not existent just a moment ago.

By telling Angular that our pipe is not pure, we can disable the optimization it performs on pure pipes. This also means that Angular will need to revalidate the output of the pipe on every change detection. This can lead to performance issues; therefore, the pure flag should be used with caution.

All right, as far as rendering tags is concerned, we are all set. Let's integrate our tags functionality into our `Editor` component so we can make use of them within the commenting

system.

Let's start by editing the `Editor` module located under `ui/editor/editor.js`:

```
...
import {TagsPipe} from '../../pipes/tags';

@Component({
  selector: 'ngc-editor',
  ...
  pipes: [TagsPipe]
})
export class Editor {
  ...
  @Input() enableTags;
  ...
}
```

First, we imported the `TagsPipe` class and referenced it to the `pipes` configuration of the `@Component` annotation.

We've also added a new input to the `enableTags` component, which will allow us to control whether we should handle tags within the content of the editor or ignore them.

That's it, as far as changes to the component file is concerned. Let's apply some changes to the template of the component by editing the `ui/editor/editor.html` file:

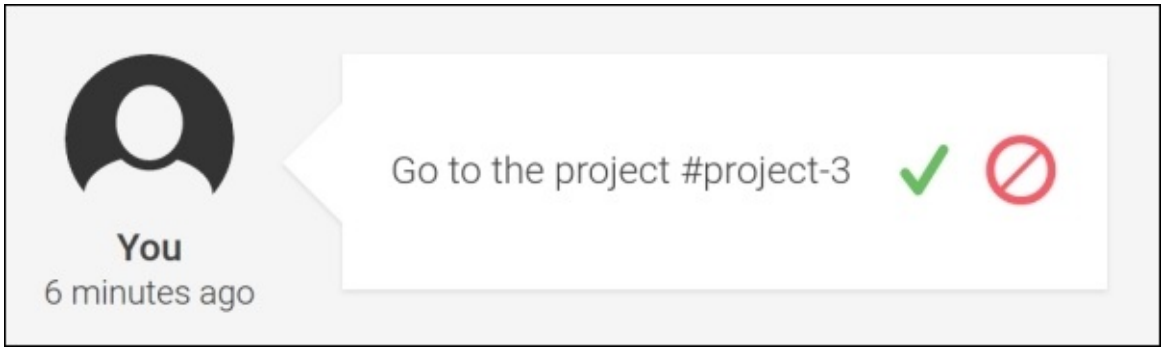
```
...
<div *ngIf="enableTags" class="editor__output"
     [innerHtml]="(content || '-') | tags"></div>
<div *ngIf="!enableTags" class="editor__output">
  {{content || '-'}}
</div>
...
```

The only change we've made in the template is where we display the editor content. We've used two template elements by employing the `NgIf` asterisk template syntax. The latter one, if tags are disabled, renders the content as before. If tags are enabled, we'll be using a property binding to the `innerHTML` property of our editor's output HTML element. This allows us to render the HTML content. In the binding, we've used our `Tags` pipe that will parse the content for tags using `TagService`.

Completion of the tags service

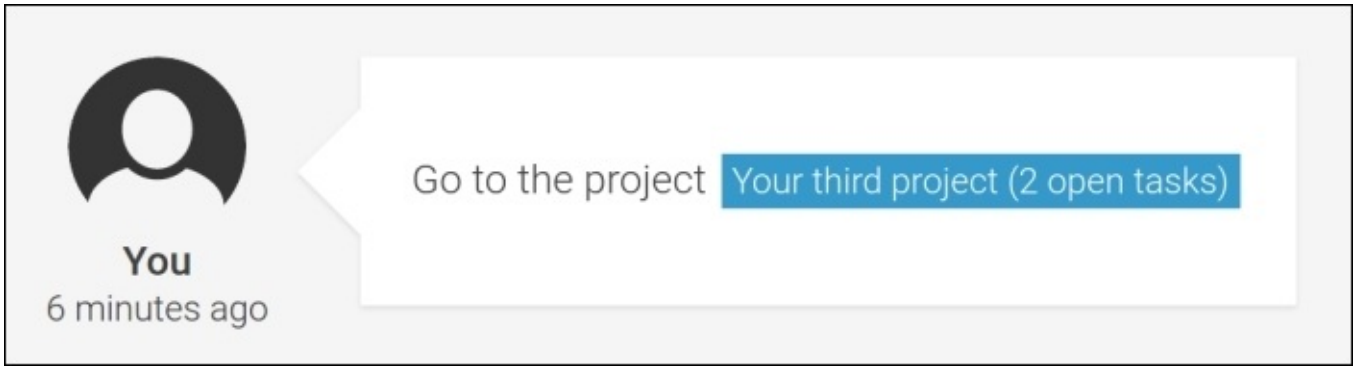
Let's digress for a moment at this point. We've already created a tagging system, and we just integrated it into our `Editor` component by using the `Tags` pipe. If a user writes project tags in any comment now, they will be rendered by `TagsService`. This is fantastic! Users can now establish cross-links to other projects within comments, which will be automatically rendered as links showing the project title and open tasks. All a user needs to do is add the text representations of project tags to a comment. In the default data set of the book, this could be the `#project-1` string.

The following two images show you an example of the commenting system. The first image is an example of an editor in edit mode, under the commenting system, where a text tag is entered:



An example where a text tag is entered

The second image is an example of a rendered tag enabled in the commenting system through our editor integration:



An example of rendered tag through editor integration

We're not done yet when it comes to entering tags. We cannot expect our users to know all the

available tags within the system and then enter them manually within comments. Let's look at how we can improve this in the next section:

In this section, we looked at the following concepts:

1. We built a tags service that generates, caches, and renders tags.
2. We built a stateful pipe using the `pure` flag.
3. We used the `[innerHTML]` property binding to render HTML content into an element.

Supporting tag input

Here, we're going to build a component and its supporting structures to make the process of entering tags a smooth experience for our users. So far, they can write project tags, but it requires them to know the project IDs, which makes our tag management quite useless. What we'd like to do is provide the user with some choices when they are about to write a tag. Ideally, we show them the available tags, as soon as they start writing a tag by typing the hash (#) symbol.

What sounds simple in the first place is actually quite a tricky thing to implement. Our tag input needs to deal with the following challenges:

- Handling input events to monitor tag creation. Somehow, we need to know when a user starts writing a tag, and we need to know when the typed tag name is updated or canceled by using an invalid tag character.
- Calculating the position of the input caret of the user. Yeah, I know this sounds pretty simple, but it actually isn't. Calculating the viewport offset position of a user's input caret requires the use of the browser's Selection API, which is quite low-level and needs some abstraction.

In order to tackle these challenges, we are going to introduce a utility class we can delegate the user input to. It will help us figure out the details we're interested in and deal with low-level APIs.

Creating a tag input manager

Create a module in a new file under `tags/tag-input-manager.js`. The first bit of code is a function that will help us figure out the position of the user input caret when the user starts typing a tag:

```
// This function can be used to find the screen coordinates of the
// input cursor position
function getRangeBoundlingClientRect() {
  const selection = window.getSelection();
  if (!selection.rangeCount) return;

  const range = selection.getRangeAt(0);

  if (!range.collapsed) {
    return range.getBoundingClientRect();
  }

  const dummy = document.createElement('span');
  range.insertNode(dummy);
  const pos = dummy.getBoundingClientRect();
  dummy.parentNode.removeChild(dummy);
  return pos;
}
```

Let's not go into too much detail here. What this code basically does is that it tries to find the bounding box `DOMRect` object, which describes the `top`, `right`, `bottom`, and `left` offsets of the caret position relative to the viewport. The problem is that the Selection API does not allow us to get the position of the caret directly; it only allows us to get the position of the current selection. In case the caret is not placed correctly, we will need to insert a dummy element at the location of the caret and return the bounding box `DOMRect` object of the dummy element. Of course, we'd need to remove the dummy element again before we return the `DOMRect` object.

Now let's create a new class, `TagInputManager`, under `lib/tags/tag-input-manager.js`, which will deal with the user input handling for tag creation:

```
export class TagInputManager {
  constructor() {
    this.reset();
  }
  ...
```

In the constructor, we need to call an internal `reset` method. This `reset` method will reset the two member fields that `TagInputManager` will expose. The `position` member will store the position of the latest caret, where the user had started writing a tag. The `textTag` member will store the current tag, which is recognized by `TagInputManager`:

```
reset() {
  this.textTag = '';
  this.position = null;
```

```
}
```

Now let's create a method to determine if a user is in the progress of entering a tag. If the `textTag` member contains a hash symbol at the beginning, we can assume that there is a tag entering in progress:

```
hasTextTag() {
  return this.textTag[0] === '#';
}
```

We also need a method that will allow us to update both the current text tag, which is entered, as well as the updated caret position:

```
updateTextTag(textTag, position = this.position) {
  this.textTag = textTag;
  this.position = position;
}
```

Within the `onKeyDown` method, we expect to receive delegated `keydown` events. We are concerned about the backspace, which should also remove the last character of the tag that is currently entered.

```
onKeyDown(event) {
  // If we receive a backspace (key code is 8), we need to
  // remove the last character from the text tag
  if (event.which === 8 && this.hasTextTag()) {
    this.updateTextTag(this.textTag.slice(0, -1));
  }
}
```

In the `onKeyPress` method, we expect to receive delegated key press events. This is where the main logic of this supporting class lies. Here, we handle two different cases:

- If the pressed key is a hash symbol, we will start over with a new tag.
- If the pressed key is not a valid word character or a hash symbol, we will reset it to its initial state, which will cancel the tag entry. Otherwise, it'd mean that we are dealing with a valid tag character, and we'll add it to the current text tag string.

The code for this is as follows:

```
onKeyPress(event) {
  const char = String.fromCharCode(event.which);
  if (char === '#') {
    // If the current character from user input is a hash symbol
    // we can initiate a new text tag and set the current
    // position
    this.updateTextTag('#', getRangeBoundlingClientRect());
  } else if ((/[\w-]/i).test(this.textTag[0])) {
    // If the current character is not a valid tag character we
    // reset our state and assume the tag entry was canceled
    this.reset();
  } else if (this.hasTextTag()) {
    // If we have any other valid tag character input, we're
```

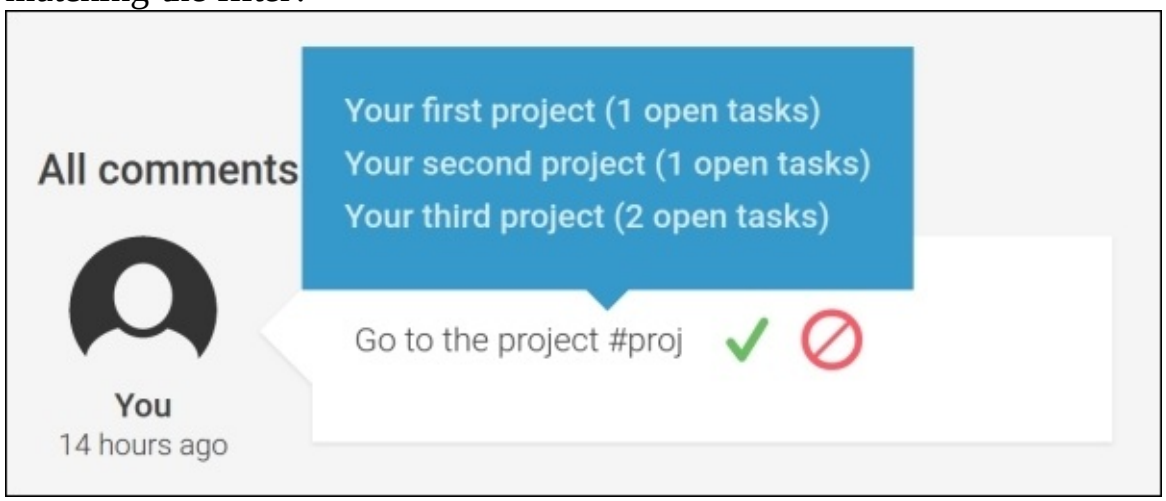
```
    // updating our text tag
    this.updateTextTag(this.textTag + char);
  }
}
```

Okay, so now we have all the support we need to handle tag input. However, we still need a way to show the available tags from `TagsService` to the user. For this purpose, we'll create a new `TagsSelect` component.

Creating a tags select component

To support the user in finding the right tag, we'll provide them with a dropdown with the available tags. To do this, we need to use our `TagInputManager` class to recognize tags within user input as well as filter the available tags with user input. Let's briefly look at the requirements of this component:

- Display the available tags gathered from `TagsService` in a tooltip/callout box
- It should support a limitation of displayed tags
- It should support an input to filter the available tags
- The component should accept an input parameter to position the callout box
- It should emit an event once the user clicks on a tag in the listed tags
- The component should hide itself if the filter is invalid or if there are no elements matching the filter:



Finished tags select component filtered with user input

Let's start with the component class and see how we fulfill these requirements. First, create a new file called `tags-select.js` under `tags/tags-select`:

```
...
@Component({
  selector: 'ngc-tags-select',
  ...
})
export class TagsSelect {
  ...
}
```

We have no specialties to deal with in our `@Component` annotation. Let's start with implementing the innards of our component. First, we'll define the following input in the component:

```
@Input() filter;
```

Using the `filter` input, we can pass a filter tag to the `TagsSelect` component. This means that we'll use the `filter` input to filter the available tags by `title` and `text` tags.

The `limit` input can be set to any number. This input is used to limit the number of filtered tags that could be displayed within the component:

```
@Input() limit;
```

The `position` input should be set to a valid `DOMRect` object that contains the top and left properties. They will be used to position our component:

```
@Input() position;
```

The `tagSelected` output property is used to emit an event once the user has clicked on a tag within the list of tags:

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

The following accessor property is bound to the host element's display style property. It will control whether the component is displayed or hidden. We only display the component if the filter is valid and the filtered tags contain at least one tag:

```
@HostBinding('style.display')
get isVisible() {
  if (this.filter[0] === '#' && this.filteredTags.length > 0) {
    return 'block';
  } else {
    return 'none';
  }
}
```

The following two accessor properties use host bindings to set the `top` and `left` styles of our host element based on the `position` input of the component:

```
@HostBinding('style.top')
get topPosition() {
  return this.position ? `${this.position.top}px` : 0;
}

@HostBinding('style.left')
get leftPosition() {
  return this.position ? `${this.position.left}px` : 0;
}
```

Let's inject `TagsService` into our component so we can access the list of available tags:

```
constructor(@Inject(TagsService) tagsService) {
  this.tagsService = tagsService;
  // This member is storing the filtered tag list
  this.filteredTags = [];
  this.filter = '';
}
```

We need to use the `OnInit` life cycle hook to set up a subscription to the `TagService` change observable. This way, we'll get access to the initial list of tags as well as any changes in the list. After we receive a new list of tags, we will need to reapply the filtering:

```
ngOnInit() {
  // The TagsService is providing us with all available tags
  // within the application
  this.tagsSubscription = this.tagsService.change.subscribe(
    (tags) => {
      // If the available tags change we store the new list and
      // execute filtering again
      this.tags = tags;
      this.filterTags();
    }
  );
}
```

The following is the method that will be called from the template if a tag is clicked. We'll just re-emit that tag using the `tagSelected` output:

```
onTagClick(tag) {
  this.tagSelected.next(tag);
}
```

The `filterTags` method is responsible for filtering and limiting our tag list based on the filter and limit input properties and the available tags from `TagsService`. As a result, it will store the filtered and limited list in the `filteredTags` member field:

```
filterTags() {
  this.filteredTags = this.tags
    .filter((tag) => {
      return tag.textTag.indexOf(this.filter.slice(1)) !== -1 ||
             tag.title.indexOf(this.filter.slice(1)) !== -1;
    })
    .slice(0, this.limit);
}
```

If the input properties filter or limit changes, we will need to reapply our filtering method. By implementing the `ngOnChanges` life cycle hook, we can easily manage this requirement:

```
ngOnChanges(changes) {
  // If the filter or the limit input changes, we're filtering the
  // available tags again
  if (this.tags && (changes.filter || changes.limit)) {
    this.filterTags();
  }
}
```

Finally, we should unsubscribe from the `TagsService` change observable if the `TagsSelect` component is destroyed:

```
ngOnDestroy() {
  this.tagsSubscription.unsubscribe();
```

```
}
```

The template for our component is rather simple. Let's look at the view template that is stored under `tags/tags-select/tags-select.html`:

```
<ul class="tags-select__list">
  <li *ngFor="let tag of filteredTags"
      (click)="onTagClick(tag)"
      class="tags-select__item">{{tag.title}}</li>
</ul>
```

We used the `NgFor` directive to iterate over all the tags within the `filteredTags` member. If a tag is clicked, we will need to execute the `onTagClicked` method and pass the tag of the current iteration. In the listing, we'll only display the tag title that should help the user identify the tag they would like to use.

Now we have built all the pieces that we need to enable smooth tag entering for our users. Let's patch our `Editor` component again to include our changes.

Integrating tag input within the editor component

As the first step, we should amend our `Editor` component to utilize the `TagInputManager` class. We need to delegate the user input inside the content-editable element to the tag input manager so it can detect any tag entering. Then, we'll use the information from `TagInputManager` to control a `TagsSelector` component.

First, let's look at the required changes to be made inside the `Component` class located under `ui/editor/editor.js`:

```
...
import {TagsSelect} from '../../tags/tags-select/tags-select';
import {TagInputManager} from '../../tags/tag-input-manager';

@Component({
  selector: 'ngc-editor',
  ...
  directives: [TagsSelect]
})
export class Editor {
  ...
    // We're using a TagInputManager to help us dealing with tag
    // creation
    this.tagInputManager = new TagInputManager();
  }
  ...
  // This method is called when the editable element receives a
  // keydown event
  onKeyDown(event) {
    // We're delegating the keydown event to the TagInputManager
    this.tagInputManager.onKeyDown(event);
  }

  // This method is called when the editable element receives a
  // keypress event
  onKeyPress(event) {
    // We're delegating the keypress event to the TagInputManager
    this.tagInputManager.onKeyPress(event);
  }

  // This method is called if the child TagSelect component is
  // emitting an event for a selected tag
  onTagSelected(tag) {
    // We replace the partial text tag within the editor with the
    // text representation of the tag that was selected in the
    // TagSelect component.
    this.setEditableContent(
      this.getEditableContent().replace(
        this.tagInputManager.textTag, tag.textTag
      )
    );
    this.tagInputManager.reset();
  }
  ...
```

```
}
```

In our `@Component` annotation, we added the `TagsSelect` component to the directives property so we could use the component within the template.

To help us do all the low-level processing for tag entry, we used `TagInputManager` and created a new instance of it within the component constructor.

We have now created two methods for handling `keypress` and `keydown` events coming from our content-editable element. These methods delegate the events to `TagInputManager`, which will handle all of the processing to extract a text tag and the position of the caret.

Finally, we added a method that will be called once a tag is clicked within the `TagsSelect` component. Here, we simply replaced the text tag that is currently entered with the text representation of the tag that was clicked. This provides a naive implementation of some sort of autocomplete. After we added the text representation of the clicked tag to the content-editable element, we reset `TagInputManager` to clear its state.

The only bit left now is to to edit the template of the `Editor` component in order to include the `TagsSelect` component.

In the `ui/editor/editor.html` file, we need to make the following changes:

```
...
<ngc-tags-select *ngIf="enableTags"
                 [filter]="tagInputManager.textTag"
                 [position]="tagInputManager.position"
                 [limit]="5"
                 (tagSelected)="onTagSelected($event)">
</ngc-tags-select>
```

The `NgIf` directive helps us avoid the component from being created if tags are not enabled within the editor.

We set the `filter` and `position` input of the `TagsSelect` component from the data we have in our `TagInputManager` instance.

On the emitted `tagSelected` event of the `TagsSelect` component, we called the `onTagSelected` method on the `Editor` component we created a moment ago.

That's all we need to do with the template of the `Editor` component.

Finishing up our tagging system

Congratulations! You've now successfully implemented the first of the three usability components.

With the help of a `TagInputManager` class, we offloaded heavy low-level handling of user input and the processing of the user caret position. Then, we created a component to display the available tags to the user and provided a way for them to select a tag by clicking on it. In our `Editor` component, we used the `TagInputManager` class together with the `TagsSelect` component to enable the smooth entering of tags while editing comments and other areas where we've enabled tagging.

We've covered the following concepts in this section:

1. We processed complex user input within a designated manager class to offload logic from our components.
2. We used host bindings to set positional style attributes.
3. We implemented fully reactive components that rely on observables and don't create side effects during change detection.

Drag and drop

We have learned to use our computer mouse and keyboard with great efficiency. Using keyboard shortcuts, different click actions and contextual mouse menus support us nicely when performing tasks. However, there is one pattern that has gained more attention again in applications lately, given the current mobile and touch devices hype. Drag and drop actions are a very intuitive and logical way to express actions such as moving or copying items. One task performed on user interfaces benefits from drag and drop particularly, which is ordering items within a list. If we need to order items via action menus, it gets very confusing. Moving items step by step using the up and down buttons works great, but it takes a lot of time. If you can drag items around and drop them in a place where you'd like them to be reordered, you can sort a list of items extremely fast.

In this topic, we will build the required elements to enable drag and drop selectively. We will use the drag and drop feature to enable users to reorder their task lists. By developing reusable directives to provide this functionality, we can enable the feature at any other spot within our application later on.

To implement our directives, we will make use of the HTML5 drag and drop API, which is supported in all the major browsers at the time of writing this book.

Since we would like to reuse our drag and drop behavior on multiple components, we will use directives for the implementation. We are going to create two directives in this section:

- **Draggable directive**: This directive should be attached to components, which should be enabled for dragging
- **Draggable drop zone directive**: This directive should be attached to components that will act as a drop target

We'll also implement a feature where we can be selective about what can be dragged where. For this, we will use a type attribute on our draggable directives as well an accepted type attribute on our drop zones.

Implementing the draggable directive

The `draggable` directive will be attached to the element that can be dragged onto other elements. Let's get started with creating a new directive class under `draggable/draggable.js`:

```
...
@Directive({
  selector: '[draggable]',
  host: {
    class: 'draggable',
    // Additionally to the class we also need to set the HTML
    // attribute draggable to enable draggable browser behavior
    draggable: 'true'
  }
})
export class Draggable {
  ...
}
```

Instead of using the `@Component` annotation, we've now used the `@Directive` annotation to let Angular know that the following class is a directive class. By setting the HTML attribute `draggable` to `true`, we tell the browser that we're considering this element a draggable element.

Tip

The big difference of using directives in comparison to components is that they don't embrace a view but only behavior. Therefore, it's also possible to use many directives on the same element, which is not possible with components.

Let's look at the input for our newly created component class:

```
@Input() draggableData;
```

The `draggableData` input is used to specify the data that represents the element which can be dragged. This data will be serialized to JSON and transferred to our drop zones once a drag action is completed.

By specifying a draggable type, we can be more selective when the element is dragged over a drop zone. Within the drop zone, we can have a counterpart that controls what types are acceptable to be dropped.

```
@Input() draggableType;
```

Additionally to our input, we also want to use a host binding to set a special class if the element is currently dragged:

```
@HostBinding('class.draggable--dragging') dragging;
```

This binding will set a `draggable--dragging` class, which will apply some special styles that

will make it easy to recognize that an element is dragged.

Now we need to handle two events within our directive to implement the behavior of a draggable element. The following DOM events are triggered by the drag and drop DOM API:

- `dragstart`: This event is emitted on elements that are grabbed and moved across the screen
- `dragend`: If the previously initiated dragging of the element is ended, because of a successful drop or a release outside of a valid drop target, this DOM event will be triggered.

Let's look at the implementation of `HostListener` for the `dragstart` event:

```
// We're listening for the dragstart event and initialize the
// dataTransfer object
@HostListener('dragstart', ['$event'])
onDragStart(event) {
  event.dataTransfer.effectAllowed = 'move';
  // Serialize our data to JSON and set it on our dataTransfer
  // object
  event.dataTransfer.setData(
    'application/json',
    JSON.stringify(this.draggableData));
  // By adding the draggableType as a data type key within our
  // The dataTransfer object, we enable drop zones to observe the type
  // before receiving the actual drop.
  event.dataTransfer.setData(

    `draggable-type:${this.draggableType}`, '');
  this.dragging = true;
}
```

Now let's discuss the different actions we will perform in the implementation of our host listener:

1. We will need to access the DOM event object in our host listener. If we were to create this binding within the template, we would probably need to write something similar to this: `(dragstart)="onDragStart($event)"`. Within event bindings, we can make use of the synthetic variable `$event`, which is a reference to the event that would have triggered the event binding. If we were to create an event binding on our host element using the `@HostListener` annotation, we would need to construct the parameter list for the binding using the second argument of the decorator.
2. The first action in our event listener is to set the desired `effectAllowed` property on the `dataTransfer` object. Currently, we only support the `move` effect as our main concern is to reorder tasks within the task list using drag and drop. The drag and drop API is very system-specific, but usually there are different drag effects if a user holds a modifier key (such as *Ctrl* or *Shift*) while initiating the dragging. Within our `draggable` directive, we can force the `move` effect for all drag actions.
3. In the next code snippet, we will set the data that should be transferred by dragging. It's important to understand the core purpose of the drag and drop API. It does not only

provide a way to implement drag and drop for elements solely in your DOM, but it also supports the dragging of files and other objects into your browser. Because of this, the API undergoes some constraints, where one of them is making it impossible to transfer data other than simple string values. In order for us to transfer complex objects, we will serialize the data from the `draggableData` input using `JSON.stringify`.

4. Another limitation caused by some security constraints within the API is that data can only be read after a successful drop. This means that we cannot inspect the data if the user is just hovering over an element. However, we need to know some facts about the data when hovering drop zones. We need to know the type of the draggable element when entering a drop zone so we can make the drop zone signal if the type is accepted. We're using a small workaround for this issue. The drag and drop API hides the data when we drag data over a drop target. However, it tells us what type of data it is. Knowing this fact, we can use the `setData` function to encode our draggable type. Accessing the data keys only is considered secure and therefore can be done in all drop zone events.
5. Finally, we'll set the dragging flag to `true`, which will cause the class binding to revalidate and add the `draggable--dragging` class to the element.

After dealing with the `dragstart` event, we only need to handle the `dragend` event to complete our `Draggable` directive. The only thing we do within the `onDragEnd` method that is bound to the `dragend` event is set the dragging member to false. This will cause the `draggable--dragging` class to be removed from the host element:

```
@HostListener('dragend')
onDragEnd() {
  this.dragging = false;
}
```

That's it for the behavior of our `Draggable` directive. Now we need to create its counterpart directive to provide the behavior of a drop zone.

Implementing a drop target directive

Drop zones will act as containers where draggable elements can be dropped. For this, we'll create a new directive called `DraggableDropZone` under `draggable/draggable-drop-zone.js`:

```
@Directive({
  selector: '[draggableDropZone]'
})
export class DraggableDropZone {
  ...
}
```

There's nothing special about this `@Directive` annotation. We used an attribute selector so it can be attached using a `draggableDropZone` attribute on any HTML element. Using the following input, we can specify what types of draggable elements we accept in this drop zone. This will help the user identify whether they are able to drop off the draggable elements when approaching the drop zone:

```
@Input() dropAcceptType;
```

Upon successful drops into the drop zone, we will need to emit an event so that the components using our drag and drop functionality can react accordingly. For this purpose, let's create a `dropDraggable` output property:

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

The `over` member field will store the state if an accepted element is in the process of being dragged over the drop zone:

```
@HostBinding('class.draggable--over') over;
```

The following method will be used to check whether our drop zone should accept any given drag and drop event by checking against our `dropAcceptType` member. If you remember the security problems we needed to work around with when creating the `Draggable` directive, you will understand why this determination is rather simple:

```
typeIsAccepted(event) {
  const draggableType =
      Array.from(event.dataTransfer.types).find(
        (key) => key.indexOf('draggable-type') === 0
      );
  return draggableType &&
         draggableType.split(':')[1] === this.dropAcceptType;
}
```

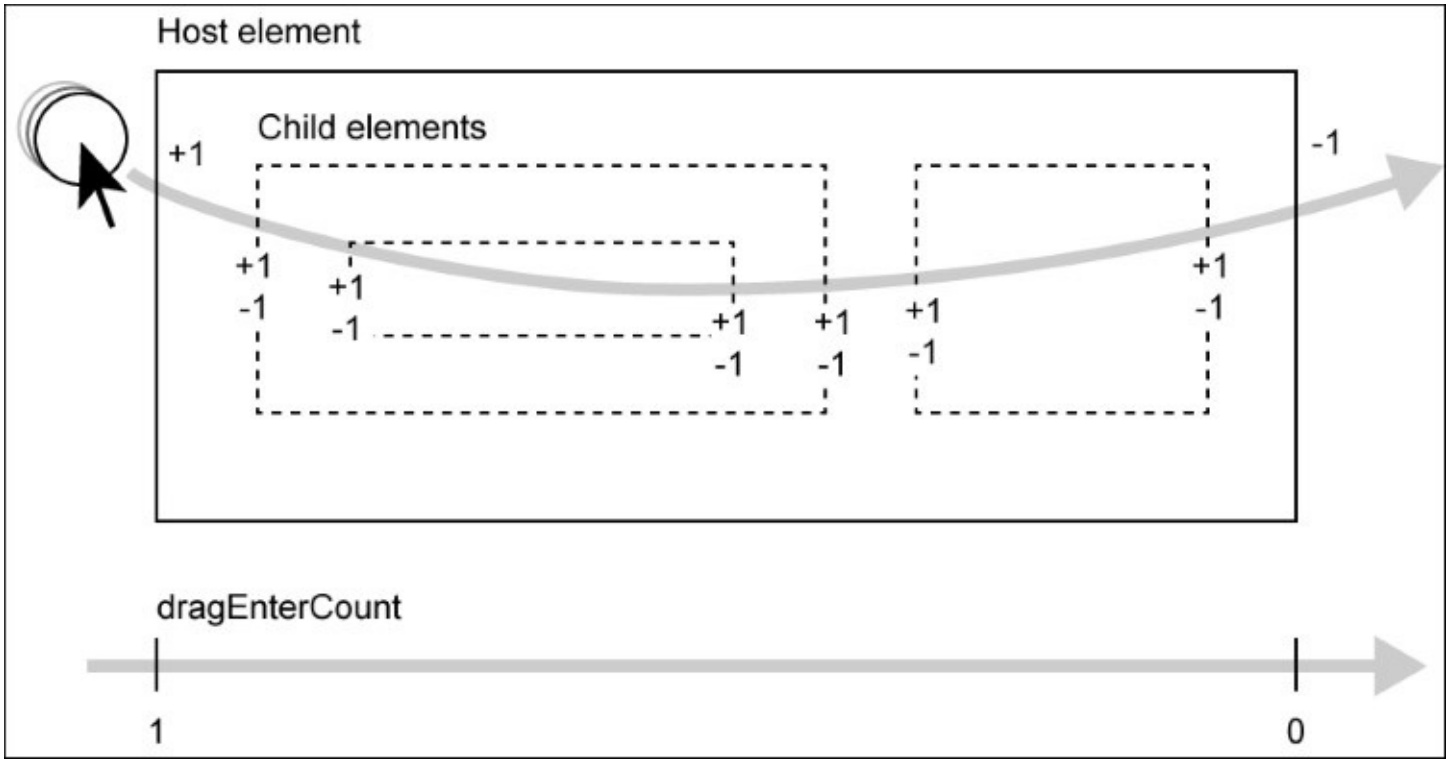
We can only read the types of the data within `dataTransfer` objects for certain events, where the data itself is hidden until a successful `drop` event is occurred. To bypass this security limitation, we've encoded the draggable type information into a data key itself. Since we can list all the data types safely, it's not too hard to extract the encoded draggable type information. We will search for a data type key that starts with `"draggable-type"` and then

split it by the column character. The value after the column character is our type information, which we will then compare against the `dropAcceptType` directive input property.

We will use two events to determine whether a draggable element is moved to our drop zone:

- `dragenter`: This is fired by an element if another element is dragged over it
- `dragleave`: This is fired by an element if the previously entered element has left again

There's one problem with the preceding events, which is that they actually bubble, and we will receive a `dragleave` event if the dragged element is moved to a child element within our drop zone. Because of the bubbling, we then also receive `dragenter` and `dragleave` events from the child elements. This is not desired in our case, and we need to build some functionality to improve this behavior. We will make use of a counter member field `dragEnterCount`, which will count up to all the `dragenter` events and count down to `dragleave` events. This way, we can now say that only on `dragleave` events, where the counter becomes zero, we will actually leave the inside of our drop zone. Let's look at the following diagram that illustrates the problem:



Visualization of important variables and functions for our calculations

Let's implement this logic to build a proper enter and leave behavior of our drop zone within the `draggable/draggable-drop-zone.js` file:

```
constructor() {
  // We need this counter to know if a draggable is still over our
  // drop zone
```

```
  this.dragEnterCount = 0;
}

// The dragenter event is captured when a draggable is dragged
// into our drop zone
@HostListener('dragenter', ['$event'])
onDragEnter(event) {
  // Only handle event if the draggable is accepted by our drop
  // zone
  if (this.typeIsAccepted(event)) {
    this.over = true;
    // We use this counter to determine if we loose focus because
    // of child element or because of final leave
    this.dragEnterCount++;
  }
}

// The dragleave event is captured when the draggable leaves our
// drop zone
@HostListener('dragleave', ['$event'])
onDragLeave(event) {
  // Using dragEnterCount, we determine if the dragleave event is
  // because of child elements or because the draggable was moved
  // outside the drop zone
  if (this.typeIsAccepted(event) && --this.dragEnterCount === 0) {
    this.over = false;
  }
}
```

Within both the events, we first check whether the event is carrying a `dataTransfer` object of which we accept the type. After validating the type using our `typeIsAccepter` method, we deal with the counter and set the `over` member field if required.

We need to handle another event that is important for drag and drop functionality, which is the `dragover` event. Within the `dragover` event, we can set the accepted `dropEffect` of the current dragging action. This will tell our browser that the initiated dragging action from our draggable is suitable for this drop zone. It's also important that we prevent the default browser behavior so there's nothing in the way of our custom drag and drop behavior. Let's add an other function to cover those concerns:

```
@HostListener('dragover', ['$event'])
onDragOver(event) {
  // Only handle event if the draggable is accepted by our drop
  // zone
  if (this.typeIsAccepted(event)) {
    // Prevent any default drag action of the browser and set the
    // dropEffect of the dataTransfer object
    event.preventDefault();
    event.dataTransfer.dropEffect = 'move';
  }
}
```

Finally, we need to handle the most important event in the drop zone, which is the `drop` event

that is triggered if a user drops a draggable into our drop zone:

```
// This event will be captured if a draggable element is dropped
// onto our drop zone
@HostListener('drop', ['$event'])
onDrop(event) {
  // Only handle event if the draggable is accepted by our drop
  // zone
  if (this.typeIsAccepted(event)) {
    // First obtain the data object that comes with the drop event
    const data = JSON.parse(
      event.dataTransfer.getData('application/json')
    );
    // After successful drop, we can reset our state and emit an
    // event with the data
    this.over = false;
    this.dragEnterCount = 0;
    this.dropDraggable.next(data);
  }
}
```

After checking whether the dropped element is of an accepted type, we can now go ahead and read the `dataTransfer` object data from the event. This data was previously set by the `Draggable` directive and needs to be deserialized using `JSON.parse`.

Since the drop was successful, we can reset our `dragEnterCount` member and set the `over` flag to `false`.

Finally, we will emit the deserialized data from the draggable element using our `dropDraggable` output property.

That's all we need to have a highly reusable drag and drop behavior that we can now attach to any components within our application where we feel the need.

Integrating drag and drop in task list component

We can now use the `Draggable` and `DraggableDropZone` directives in our `TaskList` component so we can enable the reordering of tasks using drag and drop.

The way we're going to do this is by attaching both the directives to the task elements within the `TaskList` component template, where we'll render them. Yeah, that's right! We want to make our `Task` component a draggable but also a drop zone at the same time. This way, we can drop tasks into other tasks, and this gives us the foundation for reordering. What we will do is reorder the list in a drop so that the dropped task will be squeezed into the position right before the task where it was dropped.

First, let's apply the directives to the `<ngc-task>` element in the `TaskList` component template, namely `task-list/task-list.html`:

```
<div class="task-list__l-container">
  ...
  <ngc-task *ngFor="let task of filteredTasks"
            line:[task]="task
            (taskUpdated)="onTaskUpdated(task, $event)"
            (taskDeleted)="onTaskDeleted(task)"
            draggable
            draggableType="task"
            [draggableData]="task"
            draggableDropZone
            dropAcceptType="task"
            (dropDraggable)="onTaskDrop($event, task)">
  </ngc-task>
  ...
</div>
```

Alright, using the preceding attributes, we have made our tasks not only a draggable, but also a drop zone. By specifying both `draggableType` and `dropAcceptType` to the `"task"` string, we are telling our drag and drop behavior that these task elements can be dropped into other task elements. Our `DraggableDropZone` directive is set to emit a `dropDraggable` event whenever a valid draggable is dropped off. To handle dropped tasks, we can simply use this event and create a binding to a method in our `TaskList` component.

Let's see what we need to change within our `Component` class, located under `task-list/task-list.js`, to make this work:

```
...
import {Draggable} from '../draggable/draggable';
import {DraggableDropZone} from '../draggable/draggable-drop-zone';

@Component({
  selector: 'ngc-task-list',
  ...
  directives: [..., Draggable, DraggableDropZone]
})
```

```
export class TaskList {
  ...
  onTaskDrop(source, target) {
    if (source.position === target.position) {
      return;
    }

    let tasks = this.tasks.slice();
    const sourceIndex = tasks.findIndex(
      (task) => task.position === source.position
    );
    const targetIndex = tasks.findIndex(
      (task) => task.position === target.position
    );
    tasks.splice(targetIndex,
                 0,
                 tasks.splice(sourceIndex, 1)[0]);
    tasks = tasks.map((task, index) => {
      return Object.assign({}, task, {
        position: index
      });
    });
    this.tasksUpdated.next(tasks);
  }
  ...
}
```

Let's elaborate on the behavior we'll see within the `onTaskDrop` method that is bound to the `DropZone`'s `dropDraggable` event in our template:

1. If you check the template again, you would see that we bound to the `onTaskDrop` method with the following expression: `(dropDraggable)="onTaskDrop($event, task)"`. Since the drop zone emitted an event with de-serialized data that was bound using the draggable input property `draggableData`, we can safely assume that we will receive a copy of the task that was dropped into the drop zone. As a second parameter to our binding, we added the local view variable `task`, which is actually the task that acts as the drop zone. Therefore, we can say that the first parameter of our `onTaskDrop` method represents the source, while the second represents the target task.
2. As a first check in our method, we compare the source position with the target position, and if they match, we can assume that the task was dropped by itself and we don't need to perform any further actions.
3. Now we can get the source and target task indices within our tasks array and execute a nested splice so that we can remove the source from its old position within the array and add it right before the position of the target.
4. All that's left to do now is recalculate the position fields of the tasks so that they reflect the reordered array. We can do this easily by using `Array.prototype.map`.
5. As the last step, we need to notify our parent component that we've updated the task list. We can simply use the `taskUpdated` event to do so. We have used that same event when tasks were added or removed.

How great is that? We have successfully implemented drag and drop on our task list to provide a very usable feature to reorder tasks.

Recapitulate on drag and drop

With the use of the low-level drag and drop API, using events and `dataTransfer` objects, we have implemented two directives that can now be used to execute smooth drag and drop functionality within our application wherever we desire.

With almost no effort, we have implemented our drag and drop behavior on the task list to provide a nice feature to reorder the tasks within the list. The only thing we needed to do, besides hooking up the directives, was to implement a method where we could reorder the tasks based on the information from the `DraggableDropZone` event.

We have worked with the following concepts in this section:

1. We learned the basics of HTML5 drag and drop API.
2. We used the `dataTransfer` object to securely transfer data within drag and drop events.
3. Built reusable behavior patterns using directives.
4. Enriched the standard drag and drop API by providing our own custom selection mechanisms using a custom data type that encodes draggable-type information.

To infinity and beyond!

Displaying a simple list with an average size does not come with a lot of challenges. As soon as the lists starts to grow, challenges start to appear. We can easily overwhelm a user with a very long list. Long lists can also have a performance impact on our application, especially when it displays dynamic content.

One way to address the challenge faced when displaying long lists is to provide pagination. However, pages do not always translate very well. While using pagination on a desktop device with a mouse seems very intuitive, it becomes cumbersome on mobile devices with touch support.

In this chapter, we'll look at a different approach that can help us mitigate the performance implications of long lists while providing a smooth experience on mobile devices. We are using a pattern sometimes referred to as infinite scrolling. The goal is to display only enough items within the list to fill the screen, and load more items on demand if the user scrolls down.

To implement such a behavior, we could write a wrapper component that will provide an infinite scroll pane and use content insertion to embrace our list. However, we will use a different approach to implement our infinite scroll behavior and build a custom template directive such as `NgFor`.

The asterisk syntax and templates

We have used the `NgFor` and `NgIf` directives quite a lot so far using the asterisk (`*`) symbol to indicate we're dealing with a directive that creates a template. However, we haven't looked at the anatomy of the asterisk template syntax. Imagine that it will create some sort of syntactic sugar for our template.

Check out this example of using the `NgFor` directive with the asterisk template syntax:

```
<div *ngFor="let i of [1, 2, 3]">{{i}}</div>
```

The template parser of Angular will handle all the attributes that start with an asterisk in a special way. The preceding example is an easier and more concise writing style of the following:

```
<template ngFor #i [ngForOf]="[1, 2, 3]">
  <div>{{i}}</div>
</template>
```

Both the preceding examples are absolutely identical. Template directives, such as `NgFor` or `NgIf`, make use of HTML template elements, which we've briefly discussed in Chapter 1, *Component-Based User Interfaces*. The reason that the Angular common directives `NgFor`, `NgIf`, and `NgSwitch` use HTML template elements is actually quite obvious if you think about their nature. All three directives need to insert and remove large regions of our DOM dynamically. `NgIf`, for example, inserts or removes the element it's attached to, based on a condition. By leveraging template elements, this can be supported by the browser's native functionality.

Note

If you compare the examples discussed here, it's obvious that the first writing style is much simpler to deal with. Asking you to write a separate template element every time you'd want to use `NgFor` or `NgIf` would be quite a pain. This is the only reason why the asterisk syntax exists, its raison d'être if you like. Instead of writing a template element directly, we can use an asterisk on an attribute and Angular will transform the HTML portion into a template element for us.

The `NgFor` directive uses a `TemplateRef` dependency, which can be injected into the constructor of the directive, to instantiate the template or multiple instances of it, as desired. The `[ngForOf]` property binding is generated during de-sugaring by appending the word `Of` within the `NgFor` expression to the directive name, `ngFor`. The binding is created by the `NgFor` directive, which accepts an input, `ngForOf`.

Consider the following example:

```
<div *test="let variable withSugar true">{{variable}}</div>
```

Angular would de-sugar this into the following code:

```
<template test #variable [testWithSugar]="true">
  <div>{{variable}}</div>
</template>
```

That's just the way Angular de-sugars the asterisk template syntax. It's a shortcut to attach directives as well as one input bindings to template elements.

There's still one thing that might look confusing, which is the variable attribute in the template. Let's look at another example of using the `NgFor` directive by aliasing the exposed local variables of a directive like the current index:

```
<div *ngFor="let n of [1, 2, 3]; let i = index">{{i}}: {{n}}</div>
```

This example will be de-sugared to the following template:

```
<template ngFor #n #i="index" [ngForOf]="[1, 2, 3]">
  <div>{{i}}: {{n}}</div>
</template>
```

So we can now tell from de-sugaring that additional aliases or mappings will get created as variable mappings in our template element. The index that is exposed within the code of the `NgFor` directive class as a local view variable is mapped to a local view variable within the instantiated content of the template.

So what's going on with the local view variable `n` within our instantiated templates? Why can we access `n`, when there's just one variable attribute without any value that would tell us where it's mapped to?

We have learned that when we use hash symbol attributes on regular elements, we create a local view reference. We can use this reference as an identifier in the view directly, or by querying using `@ViewChild`. However, when the view compiler of Angular discovers what looks like a local view reference on a template, the behavior is a bit different.

What is invisible to us is that Angular actually implies a default value for variable attributes on template elements that don't have an attribute value. It will create a mapping for a local view variable called `$implicit`. You can think of `$implicit` as a default value that can be exposed in directives as local view variables and will provide some ease of use when dealing with template elements.

It would also be totally valid to write the preceding example as follows:

```
<template ngFor #n="$implicit" #i="index" [ngForOf]="[1, 2, 3]">
  <div>{{i}}: {{n}}</div>
</template>
```

Here, the `NgFor` directive is exposing a local view variable `$implicit`, which is a reference to the current value associated with the instance during the iteration over the array it receives

within the `ngForOf` input. Using a plain variable attribute without a value, Angular will default a mapping to `$implicit`. Because we don't want to write this mapping all the time ourselves, we can just specify an empty variable attribute and Angular will assume it for us.

Creating an infinite scroll directive

Since we now know a bit more about template elements and how Angular deals with the asterisk syntax, we can actually create our own copy of `NgFor`, which is additionally dealing with the behavior of infinite scrolling.

Let's create a new file for our directive under `infinite-scroll/infinite-scroll.js`:

```
...
@Directive({
  selector: '[ngcInfiniteScroll]'
})
export class InfiniteScroll {
  ...
  // This input will be set by the for of template syntax
  @Input('ngcInfiniteScrollOf')
  set infiniteScrollOfSetter(value) {
    this.infiniteScrollOf = value;
    ...
  }
  ...

  applyChanges(changes) {
    ...
    this.bulkInsert(insertTuples).forEach((tuple) =>
      tuple.view.context.$implicit = tuple.record.item);
  }
  ...
}
```

We start off by declaring a regular directive that is sensitive to the attribute selector `ngcInfiniteScroll`. The preceding code excerpt only shows the relevant code for the template element handling we discussed in the previous topic. There are some code parts that we will cover later on in this topic. You can see that we used an input property `ngcInfiniteScrollOf`, which is used to pass in the list of times used within the infinite scrolling. For inserted template instances, we set the local view variable `$implicit` to the actual item within the list we were iterating over.

We will discuss how we get to all of the surrounding code shortly, but first let's take a look at how we could use this directive within a template:

```
<div *ngcInfiniteScroll="#item of items">{{item}}</div>
```

The preceding code, as per the mechanisms described in the previous topic, will de-sugar into the following template element:

```
<template ngcInfiniteScroll #item [ngcInfiniteScrollOf]="items">
  <div>{{item}}</div>
</template>
```

So what we can tell now is that the items array will be placed as a property binding onto our

template element. The same element also contains the `InfiniteScroll` directive.

After discussing how the directive will be used and how we can get the required input into the directive, let's look at the implementation details that enable the infinite scroll behavior.

Our directive needs to deal with quite a lot of concerns. Let's check out a high-level requirement list:

- It needs to dynamically create new child views based on the template element and also remove child views that are no longer required.
- It needs to detect changes on the input property `ngcInfiniteScrollOf`, which is bound to an array within the template. It's not sufficient to use a simple identity check, because we'd like to create a comparison of the previous array to the new array and only perform view changes based on the differences. For this purpose, we will need to implement the `DoCheck` life cycle callback.
- It needs to store a count of items that should be displayed initially, and by detecting scroll events, the displayed item count should increase so more items could be made visible. At the same time, scrolling should trigger the change detection so that we can create new instances of the template within the view.

Let's start with the constructor of our directive:

```
constructor(@Inject(ViewContainerRef) viewContainerRef,
            @Inject(TemplateRef) templateRef,
            @Inject(IterableDiffers) iterableDiffers,
            @Inject(ChangeDetectorRef) cdr) {
  // Using Object.assign we can easily add all constructor
  // arguments to our instance
  Object.assign(this,
    {viewContainerRef, templateRef, iterableDiffers, cdr});
  // How many items will be shown initially
  this.shownItemCount = 3;
  // How many items should be displayed additionally, when we
  // scroll to the bottom
  this.increment = 3;
}
```

We needed to use quite a lot of injected dependencies in order to perform all the operations required to fulfill the outlined requirements of our directive:

- The `ViewContainerRef` dependency helps us create new embedded views based on our template element as well as detach or completely remove existing views.
- The `TemplateRef` dependency is a reference to the template element, and we can use it in conjunction with the `ViewContainerRef` dependency in order to create new instances.
- The `IterableDiffers` dependency is used to create a diff of our input property, which is the array of items we're concerned about in our infinite scroll repeater. It supports us in finding the created, removed, and deleted items.
- The `ChangeDetectorRef` dependency is used to trigger change detection manually when we actually need it.

As the first step, we used `Object.assign` to store all our function parameters in the instance of the directive. Then, we set two-member variables that will store information related to the number of items that should be displayed and also the number of displayed items we should increase upon scrolling.

That's it for the constructor. We also need to perform some actions after the view within our directive has been initialized. We'll use the `ngOnInit` life cycle hook for this purpose:

```
ngOnInit() {
  this.scrollableElement = findScrollableParent(
    this.viewContainerRef.element.nativeElement.parentElement);
  this.scrollableElement.addEventListener('scroll', this._onScrollListener);
}
```

Let's look at these two lines of code in more detail:

- The way our infinite scrolling works is that it detects whether the scrollable parent element has already scrolled to the bottom. If that's the case, we'd need to render more items from the list. In order to check whether our parent element has already scrolled to the bottom, we will need a reference to it. As scroll events don't bubble, we need to be very precise where to monitor them. That's the reason why we use a utility function to scan the DOM tree to find the next scrollable parent element. The `findScrollableParent` function looks for the first parent element that has scrollbars or the window object. You can check the source code of this chapter if you'd like to see the internals of the function.
- Now we've added an event handler to the found scrollable parent element and registered our internal `onScroll` method as a callback.

Detecting change within our template directive

Now let's look at the complete code of the `ngcInfiniteScrollOf` property setter, which we have briefly looked at already:

```
@Input('ngcInfiniteScrollOf')
set infiniteScrollOfSetter(value) {
  this.infiniteScrollOf = value;
  // Create a new iterable differ for the iterable `value`, if the
  // differ is not already present
  if (value && !this.differ) {
    this.differ = this.iterableDiffers.find(value).create(this.cdr);
    }
}
```

Our property setter will be called by Angular every time the `ngcInfiniteScrollOf` input property changes. Since this property is bound by the de-sugaring of the asterisk template syntax to the list we refer to within our template, we can assume that the value will always be an array or a similar iterable structure.

Besides storing the new value from the input property onto our directive instance, we also lazy initialize a member field called `differ`. Using the `find` method on the `IterableDiffers` object, we can obtain a factory that matches the type of iterable you're dealing with (in our case, this will be plain arrays). On the obtained factory, we can then call the `create` method to create a new differ. The `create` method expects a `ChangeDetectorRef` object to be passed. Luckily, we have that readily available through an injection within the constructor.

The differ will help us in a later step to detect changes between the existing value of our array and an updated one. We can then perform additions, removals, and movements in a very performant way.

If we call the `diff` method on `IterableDiffer`, it will return a new `IterableDiffer` object that contains all the changes relative to the previous `IterableDiffer` object. In a differ, we can then call one of the following methods to iterate over the relevant `CollectionChangeRecord`:

- `forEachItem`: This iterates over each `CollectionChangeRecord` within the differ by providing a callback function. The first argument to the callback will be a change record.
- `forEachPreviousItem`: This only iterates over each `CollectionChangeRecord` within the differ that already existed in the previous differ.
- `forEachAddedItem`: This only iterates over each change record that was added from the previous diff to the current one.
- `forEachMovedItem`: This only iterates over each change record that was moved.
- `forEachRemovedItem`: This only iterates over change records that were removed

The `CollectionChangeRecord` objects contain the following three main properties:

- `item`: A reference to the item within the list which we're observing for changes using the

differ
- `previousIndex`: The index of the item within the list before the `differ` iterable happened
- `currentIndex`: The index of the item within the list after the `differ` iterable

We can also solely tell from the constellation of `previousIndex` and `currentIndex` what happened to the item. The following methods are present on an `IterableDiffer` object:

- **Added items**: This can be identified if `previousIndex` is null and `currentIndex` is set to a valid number
- **Moved items**: This can be identified if `previousIndex` and `currentIndex` are both set to a valid number
- **Removed items**: This can be identified if `previousIndex` is set to a valid number but `currentIndex` is set to `null`

Now let's look at the `onScroll` method, which will be invoked by the scroll event callback of the scrollable container element. In this method, we need to handle the logic of our behavior that should be executed when a user scrolls down:

```
onScroll() {
  // If the scrollable parent is scrolled to the bottom, we will
  // increase the count of displayed items
  if (this.scrollableElement && isScrolledBottom(this.  scrollableElement)) {
    this.shownItemCount = Math.min(this.infiniteScrollOf.length,
this.shownItemCount + this.increment);
    // After incrementing the number of items displayed, we need
    // to tell the change detection to revalidate
    this.cdr.markForCheck();
  }
}
```

In the `onScroll` method, we first checked whether the scrollbar of the scrollable parent element has already scrolled to the bottom. If that's the case, we can assume that we should display more items from our list.

We incremented the `showItemCount` member by the default `increment` value, which we have set to `3`, and after modifying the number of displayed items, we used the change detector to mark our subtree structure to be checked.

Since we would like to use the differ that we have lazy initialized within our input setter to detect changes and perform any actions manually, we will need to implement the `DoCheck` life cycle callback on our directive. By implementing this, we will disable the default change detection of Angular and implement our own way to deal with changes:

```
ngDoCheck() {
  if (this.differ) {
    // We are creating a new slice based on the displayed item
    // count and then create a changes object containing the
    // differences using the IterableDiffer
    const updatedList = this.infiniteScrollOf
      .slice(0, this.shownItemCount);
```

```
    const changes = this.differ.diff(updatedList);
    if (changes) {
      // If we have any changes, we call our `applyChanges` method
      this.applyChanges(changes);
    }
  }
}
```

First, we used the differ to obtain a change record set from the current `infiniteScrollOf` array to the previous one. The differ will actually always store the previous value, so we only need to pass it the current value. The change records will then help us to perform different actions for added, removed, and moved items. It's also important to note that we did not use the whole list here to create a diff, but a slice of the list where our `showItemCount` member comes into play. This will only make the list that we're concerned about available in our infinite scroll behavior.

Adding and removing embedded views

If there are any changes detected by the differ, we can call the `applyChanges` method, which deals with the details of how to perform view updates with changed items:

```
applyChanges(changes) {
  // First we create a record list that contains all moved and
  // removed change records
  const recordViewTuples = [];
  changes.forEachRemovedItem((removedRecord) =>
    recordViewTuples.push({record: removedRecord}));
  changes.forEachMovedItem((movedRecord) =>
    recordViewTuples.push({record: movedRecord}));

  // We can now bulk remove all moved and removed views and as a
  // result we get all moved records only
  const insertTuples = this.bulkRemove(recordViewTuples);
  // In addition to all moved records we also add a record for all
  // newly added records
  changes.forEachAddedItem((addedRecord) =>
    insertTuples.push({record: addedRecord}));

  // Now we have stored all moved and added records within `
  // insertTuples` which we use to do a bulk insert. As a result
  // we get the list of the newly created views. On those views
  // we're then creating a view local variable `$implicit` that
  // will bind the list items to the variable name used within the
  // for of template syntax.
  this.bulkInsert(insertTuples).forEach((tuple) =>
    tuple.view.context.$implicit = tuple.record.item);
}
```

Let's look at the innards of the `applyChanges` method. It needs to be called from the `OnChange` life cycle hook with a parameter that reflects the record changes within the observed input array called `infiniteScrollOf`. In the constant `recordViewTuples`, we stored all the change records that were either moved or removed completely. Now you can call the `bulkRemove` method by passing the `recordViewTuples` array. The `bulkRemove` method will either detach the view, in case there is movement, or completely remove the view. The returned value is a list that will contain only the tuples that were moved. We stored these within a constant called `insertTuples`. Because they were detached from the view container, we will need to reattach them at a different position within the view container.

Now we can go ahead and add all the records to the `insertTuples` array that were added according to the latest diff. The `insertTuples` array now contains all the moved as well as added records.

Using this list, we call the `bulkInsert` method, which will reinsert moved views and create new embedded views for added records. As a result, we get a list of all the inserted records (moved and added), where each record also contains a view property that points to the inserted view.

The last step within our `applyChanges` method should now ring a bell. We iterated through the list of newly inserted views and set the local view variable `$implicit` on the view context. This way, we can set the required variable that is used to create the default variable mappings on our template elements, as discussed in the previous topic.

In order to understand how we can instantiate new views from our template element, move views around, and remove existing views, we need to understand the view container. The `ViewContainerRef` dependency is provided to our directive or component using injection in the constructor. It stores a list of views and provides some methods to add new and remove existing views. Each component within Angular contains one view container. We can then access the methods on the view container in order to programmatically modify the view.

There are four main methods in `ViewContainerRef` that we're interested in:

Method	**Description**
`createEmbeddedView`	This method will create a new embedded view using a template reference and insert the newly created view at a given index within the view container. Embedded views are views instantiated from template elements. The following are its parameters: • `templateRef`: The first parameter should be the template reference, which should be instantiated into an embedded view. • `context`: This is an optional context object, which will be available for the instantiated template view. All properties within the context can be used within the view template as local view variables. • `index`: The optional index parameter can be used to place the instantiated view at a given position within the view container. This method returns the created embedded view.
`detach`	The `detach` method will remove an embedded view from the view container at a given index without destroying the view so it can be reattached later using the `insert` method. The following is its parameter: • `index`: This is the index of the embedded view, which should be detached This method returns the detached embedded view.

`remove`	The `remove` method will completely remove an embedded view from the view container and also destroy the view. A view that has been destroyed can't simply be reattached using the `insert` method. The following is its parameter: • `index`: This is the index of the embedded view, which should be removed This method returns the removed embedded view.
`insert`	This method will insert an existing view into the view container. The following are its parameters: • `viewRef`: The embedded view that should be inserted into the view container. • `index`: The optional index parameter that can be used to place the embedded view at a given position within the view container. This method returns the inserted embedded view.

Let's quickly look at the `bulkRemove` and `bulkInsert` methods to see how we can use the view container to modify the containing view upon changes:

```
bulkRemove(tuples) {
  ...
  // Reducing the change records so we can return only moved
  // records
  return tuples.reduceRight((movedTuples, tuple) => {
    // If an index is present on the change record, it means that
    // its of type "moved"
    if (tuple.record.currentIndex != null) {
      // For moved records we only detach the view from the view
      // container and push it into the reduced record list
      tuple.view = this.viewContainerRef.detach(tuple.record.previousIndex);
      movedTuples.push(tuple);
    } else {
      // If we're dealing with a record of type "removed", we
      // completely remove the view
      this.viewContainerRef.remove(tuple.record.previousIndex);
    }
    return movedTuples;
  }, []);
}
```

We use `ViewContainerRef` to detach views in case the record contains a valid `currentIndex` field. If that's the case, we know that we are dealing with a view that will be moved. We use the `detach` method to exclude the view from its position within the view container, but this will

not destroy the view. It's important to note here that we stored the returned view from the `detach` method onto the tuple before we added it to the `movedTuples` list. This way, we were able to identify it later as a moved item, and we could use the view to reattach it using the `insert` method on the view container.

In the case where there's no valid `currentIndex`, we are dealing with an element that was removed from the list. In such cases, we'd need to use the `remove` method to completely destroy the view and remove it from the view container.

Now we'll call the `bulkInsert` method with any moved or inserted views. Let's also look at the code of this method briefly to see how we can handle view updates there:

```
bulkInsert(tuples) {
  ...
  tuples.forEach((tuple) => {
    if (tuple.view) {
      // We're inserting back the detached view at the new positionwithin the
view container
      this.viewContainerRef.insert(tuple.view,
        tuple.record.currentIndex);
    } else {
      // We're dealing with a newly created view so we create a new embedded
view on the view container and store it in the change record
      tuple.view =
        this.viewContainerRef.createEmbeddedView(
          this.templateRef,
          {},
          tuple.record.currentIndex);
    }
  });
  return tuples;
}
```

If the tuple contains a `view` property, we know we have previously detached it from a different position. We are using the insert method of the view container to reattach it at the new position using the information from `CollectionChangeRecord`.

If there's no `view` property, we are dealing with a newly added record. In that case, we simply use the `createEmbeddedView` method to create a new template instance. For the context parameter, we need to pass a new empty object. However, we've updated the context object already within our `applyChanges` method. There, we added the `$implicit` local view variable for every created view.

That's all we need for our `InfiniteScroll` directive, and we can now add it to the templates where we're planning to use this functionality. Let's use it within the task list by adding the directive to the directive list of the `TaskList` component within the `task-list/task-list.js` file:

```
...
import {InfiniteScroll} from '../infinite-scroll/infinite-scroll';
```

```
@Component({
  selector: 'ngc-task-list',
  ...
  directives: [..., InfiniteScroll]
})
export class TaskList {
  ...
}
```

Now we can simply edit the task list template in `task-list/task-list.html` and replace the `NgFor` directive with our `InfiniteScroll` directive:

```
<ngc-task *ngcInfiniteScroll="let task of filteredTasks"
          [[task]="task"
          (taskUpdated)="onTaskUpdated(task, $event)"
          (taskDeleted)="onTaskDeleted(task)"
          draggable
          draggableType="task"
          [draggableData]="task"
          draggableDropZone
          dropAcceptType="task"
          (dropDraggable)="onTaskDrop($event, task)"></ngc-task>
```

That's all we need to use our infinite scroll functionality. This is highly reusable, and we can place it wherever we'd like to use it instead of the regular `NgFor` repeater.

Finishing our infinite scroll directive

In this topic, we created an infinite scrolling behavior by implementing a template directive similar to `NgFor`. We replaced the `NgFor` directive in our task list to use the `InfiniteScroll` directive instead. Now we don't display all the tasks right at the beginning, but as soon as the user starts to scroll, new tasks appear. In scenarios where we rely on a list that is partially loaded from the server, our directive could even be extended so it could request for more items from the server on demand.

We've covered the following subtopics here:

- The asterisk syntax and de-sugaring to template elements
- The local view variable, `$implicit`
- Implementing the `OnChange` life cycle hook to provide custom change detection
- Using `IterableDiffer` to analyze the difference of changes within our array input property and handling `CollectionChangeRecord` objects to react on changes
- Using `ViewContainerRef` to update the view of a component programmatically
- Using `TemplateRef` as a reference to the template element within template directives

Summary

In this chapter, we built three components to enhance the usability of our application. Users can now make use of tags to easily annotate comments with navigable items that provide summaries to the subject. They can use drag and drop to reorder tasks and benefit from an infinite scroll behavior on the task list.

Usability is a key asset in today's applications, and by providing highly encapsulated and reusable components to address usability concerns, we can make our lives a lot easier when building those applications. Thinking in terms of components when dealing with usability is a very good thing, which not only eases development, but also establishes consistency. The consistency itself then plays a major role in making an application usable.

In the next chapter, we're going to create some nifty components to manage time within our task management system. This will also include some new user input components to enable simple work time-entry fields.

Chapter 8. Time Will Tell

Our task-management system is coming into shape. However, we were not concerned about one crucial aspect of managing our projects so far. Time plays a major role in all projects, and this is the thing that is often the most complicated to manage.

In this chapter, we will add a few features to our task management system that will help our users to manage time more efficiently. Reusing some components that we created earlier, we will be able to provide a consistent user experience to manage time.

On a higher level, we will develop the following features to enable time management in our application:

- **Task details**: So far, we did not include a details page of tasks because all the necessary information about tasks could be displayed on the task list of our project page. While our time management will increase the complexity of our tasks quite a bit, we will create a new detail view of project tasks that will also be accessible through routing.
- **Efforts management**: We will include some new data on our tasks to manage efforts on tasks. Efforts are always represented by an estimated duration of time and an effective duration of spent time. We will make both properties of efforts optional so that they can exist independently. We will create new components to enable users to provide time duration input easily.
- **Milestone management**: We will include a way to manage project milestones and then map them to project tasks. This will help us later gain an overview over the project status, and it enables the user to group tasks into smaller chunks of work.

The following topics will be covered in this chapter:

- Creating a project task detail component to edit task details and enable a new route
- Modifying our tag management system to include task tags
- Creating new pipes to deal with formatting time durations
- Creating task information components to display task overview information on the existing task components
- Creating a time duration use input component that enables users to easily input time durations
- Creating an SVG component to display progress on tasks
- Creating an autocomplete component to manage milestones on tasks

Task details

So far, our task list was sufficient enough to display all details of tasks directly in the listing. However, as we will add more details to tasks in this chapter, it's time to provide a detail view where users can edit the task.

We already laid the groundwork on project navigation using the router in Chapter 5, *Component-Based Routing*, of this book. Adding a new routable component that we'll use in the context of our projects will be a breeze.

Let's create a new component class for our project task detail view in the `project/project-task-details/project-task-details.js` path:

```
…
@Component({
  selector: 'ngc-project-task-details',
  …
})
export class ProjectTaskDetails {
  …
}
```

As this component will never exist without a parent `Project` component, we can safely rely on that to obtain the data we use. This component isn't used in pure UI composition cases, so it's not required to create a routable wrapper component like we did for other components in Chapter 5, *Component-Based Routing*. We can directly rely on route parameters and obtain the relevant data from the parent `Project` component.

First, we use dependency injection in order to get a reference to the parent project component:

```
constructor(@Inject(forwardRef(() => Project)) project) {
  this.project = project;
}
```

Similarly to our routing wrapper components, we make use of parent component injection to obtain a reference to the parent `Project` component.

Now, we'll use the `OnActivate` lifecycle hook of the router again to obtain the task number from the active route segment:

```
  routerOnActivate(currentRouteSegment) {
    const taskNr = currentRouteSegment.getParam('nr');
    this.projectChangeSubscription =
this.project.document.change.subscribe((data) => {
      this.task = data.tasks.find((task) => task.nr === +taskNr);
      this.projectMilestones = data.milestones || [];
    });
  }
```

Finally, we'll create a reactive subscription to the `LiveDocument` projects that will extract the task that we are concerned about and store it into the components `task` member. In this way, we ensure that our component will always receive the latest task data when the project is updated outside of the current task details view.

If our component gets destroyed, we need to make sure that we unsubscribe from the RxJS `Observable` that is provided by the `LiveDocument` project. Let's implement the `ngOnDestroy` lifecycle hook for this purpose:

```
ngOnDestroy() {
  this.projectChangeSubscription.unsubscribe();
}
```

Alright, let's now take a look at the template of our component, and see how we'll deal with the task data to provide an interface to edit the details. We'll create a `project-task-details.html` file in our new `component` folder:

```
<h3 class="task-details__title">
  Task Details of task #{{task?.nr}}
</h3>
<div class="task-details__content">
  <div class="task-details__label">Title</div>
  <ngc-editor [content]="task?.title"
              [showControls]="true"
              (editSaved)="onTitleSaved($event)"></ngc-editor>
  <div class="task-details__label">Description</div>
  <ngc-editor [content]="task?.description"
              [showControls]="true"
              [enableTags]="true"
              (editSaved)="onDescriptionSaved($event)">
  </ngc-editor>
</div>
```

Reusing the `Editor` component that we created in Chapter 4, *No Comments, Please!*, of this book, we can rely on simple UI composition to make the title and description of our tasks editable.

As we stored the task data into the `task` member variable on our component, we can reference the `title` and `description` fields to create a binding to the `content` input property of our editor components.

While the `title` should only consist of plaintext, we can support the tagging functionality that we created in Chapter 7, *Components for User Experience*, on the `description` field of the task. For this, we simply set the `enableTags` input property of the description `Editor` component to `true`.

The `Editor` component has an `editSaved` output property that will emit the updated content once a user saves his edits. Now, all we need to make sure of is that we create a binding to our component that will persist these changes. Let's create the `onTitleSaved` and

`onDescriptionSaved` methods on our `Component` class to handle these events:

```
onTitleSaved(title) {
  this.task.title = title;
  this.project.document.persist();
}

onDescriptionSaved(description) {
  this.task.description = description;
  this.project.document.persist();
}
```

The task member is just a reference to the given task in the `LiveDocument` project of the `Project` component. This simplifies the way we persist the data that was changed on the task. After updating the given property on the task, we simply call the `persist` method on the `LiveDocument` projects to store our changes in the data store.

So far, so good. We created a task details component that makes it easy to edit the title and description of tasks using our `Editor` UI component. The only thing left to enable our component is to create a child route on the `Project` component. Let's open our `Project` component class in `lib/project/project.js` to make the necessary modifications:

```
…
import {ProjectTaskDetails} from './project-task-details/project-task-
details';

…
@Component({
  selector: 'ngc-project',
  …
})
@Routes([
  new Route({ path: 'task/:nr', component: ProjectTaskDetails}),
  …
])
export class Project {
  …
}
```

We added a new child route on our `Project` component, which is responsible for the instantiation of our `ProjectTaskDetails` component. By including a `:nr` parameter in the route configuration, we can pass the concerned task number into the `ProjectTaskDetails` component.

Our newly-created child route is now accessible in the router and we can access the task detail view using the `/projects/project-1/task/1` example URL.

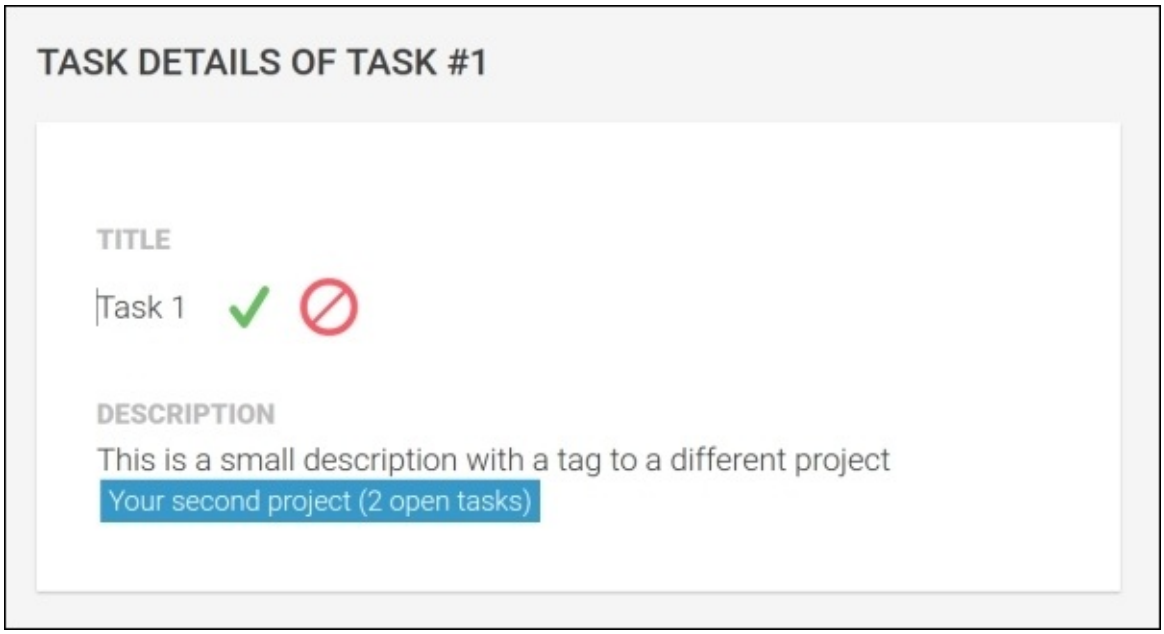
In order to make our `TaskDetails` route navigable, we need to add a navigation link to our `Task` component so that users can navigate to it in the projects task list.

For this rather simple task, the only thing that we need to do is use the `RouterLink` directive

and create a new link in the `Task` template, `lib/task-list/task/task.html`:

```
…
<div class="task__l-box-b">
  …
  <a [routerLink]="['../task', task?.nr]"
          class="button button--small">Details</a>
</div>
…
```

We use a relative router URL here because we're already on the `/project/tasks` route. As our `task/:nr` route is part of the project router, we need to navigate one level back to access the `task` route:



Newly created task detail view with editable title and description

Enabling tags for tasks

So far, the tag-management system that we created in Chapter 7, *Components for User Experience*, only supports project tags. As we now created a detail view to tasks, it would be nice to also support task tags directly in our tagging system. Our tagging system is quite flexible, and we can implement new tags with very little effort. On a higher level, we need to make the following changes to enable task tags in our system:

- Edit the `generate-tag.js` module in order to support the generation of task tags from task and project data
- Edit the `TagsService` in order to initialize task tags using the `generate-tag.js` module and cache

Let's first modify the `lib/tags/generate-tag.js` file to enable task tag generation:

```
…
export const TAG_TYPE_TASK = 'task';

export function generateTag(subject) {
  if (subject.type === TAG_TYPE_PROJECT) {
    …
  } else if (subject.type === TAG_TYPE_TASK) {
    // If we're dealing with a task, we generate the according tag
    // object
    return new Tag(
      `#${subject.project._id}-task-${subject.task.nr}`,
      `${limitWithEllipsis(subject.task.title, 20)} (${subject.task.done ?
'done' : 'open'})`,
      `#/projects/${subject.project._id}/task/${subject.task.nr}`,
      TAG_TYPE_TASK
    );
  }
}
```

As we need to have a reference to project data as well as to the individual task of this project, we expect the `subject` parameter to look like the following object:

```
{task: …, project: …, type: TAG_TYPE_TASK}
```

From this `subject` object, we can then create a new `Tag` object. For the `textTag` field, we use a construct that includes the project ID as well as the task number. Like this, we can uniquely identify the task using a simple text representation.

For the `link` field, we construct a URL from the project as well as the task number. This string will resolve to a URL required to activate the `TaskDetails` route, which we configured in the previous section.

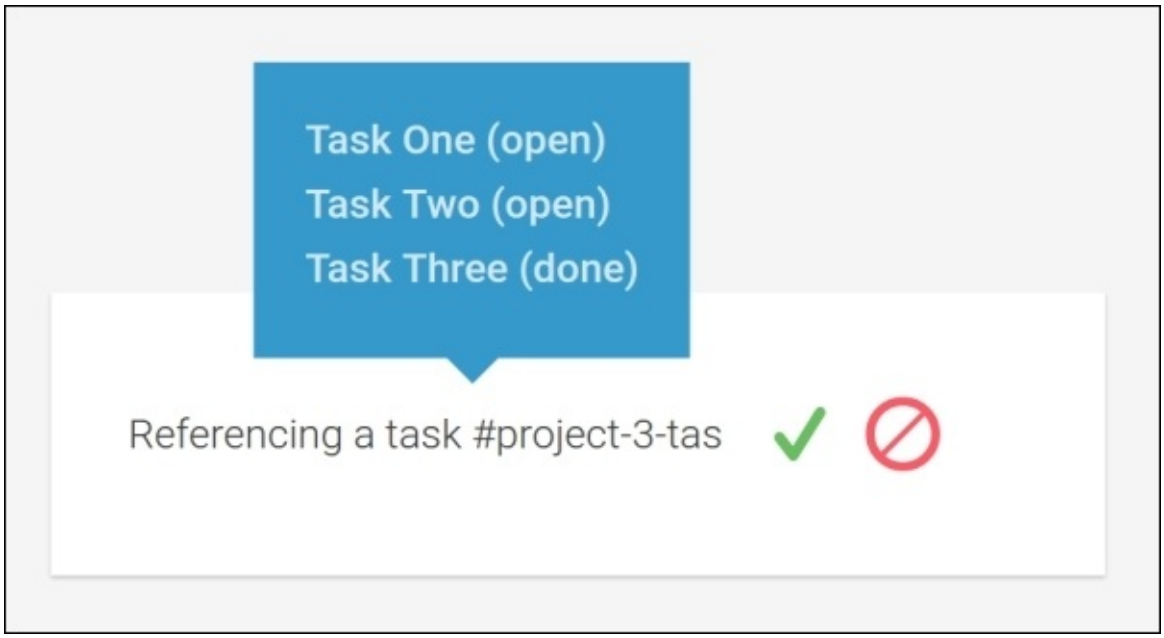
Our `generateTag` function is now ready to create task tags. Now, the only thing left to enable task tags in our system is the modification required in the `TagsService` class. Let's open the

`lib/tags/tags-service.js` file and apply our changes:

```
…
import {generateTag, TAG_TYPE_TASK} from './generate-tag';
…
@Injectable()
export class TagsService {
  …
  // This method is used internally to initialize all available
  // tags
  initializeTags() {
    …
    // Let's also create task tags
    this.projects.forEach((project) => {
      this.tags = this.tags.concat(project.tasks.map((task) => {
        return {
          type: TAG_TYPE_TASK,
          project,
          task
        };
      }).map(generateTag));
    });
    …
  }
  …
}
```

In the `initializeTags` method of our `TagsService` class, we now add task `Tag` objects for all available tasks in projects. First, we map each project task to the required `subject` object by the `generateTag` function. Then, we can simply map the resulting array using the `generateTag` function directly. The result is an array of generated task `Tag` objects that we then concatenate into the `tags` list of the `TagsService` class.

This wasn't too complicated, right? This relatively simple change results in a huge improvement for our users. They can now reference individual tasks everywhere in our system where we enabled tags:



The Editor component displaying newly-added task tags

Managing efforts

In this section, we will create some components that help us keep track of efforts. Primarily, we will use this to manage efforts on tasks, but this could be applied to any part of our application where we need to keep track of time.

Efforts in our context always consist of two components:

- **Estimated duration**: This is the duration that is initially estimated for the task
- **Effective duration**: This is the duration of time that is spent on a given task

For time durations, we assume some time units and rules that will simplify the processing of time and align to some working standards. The goal here is not to provide a razor sharp time management but something that is accurate enough to bring value. For this purpose, we define the following working time units:

- **Minute**: One minute is a regular 60 seconds
- **Hour**: One hour always represents 60 minutes
- **Day**: One day represents a regular workday of eight hours
- **Week**: One week is equivalent to five working days (5 * 8 hours)

The time duration input

We can now start to write a complex user interface component, where users can enter individual time units in different input element. However, I believe it's much more convenient to treat time duration input with a no-UI approach. Therefore, instead of building a complex user interface, we can simply agree on a textual short form to write durations, and let the user write something, such as `1.5d` or `5h 30m`, in order to provide input. Sticking to the convention that we previously established, we can build a simple parser that can handle this sort of input.

This approach has several advantages. Besides that, this is one of the most effective ways to enter time durations, and it's also easy for us to implement. We can simply reuse our `Editor` component to gather text input from the user. Then, we use a conversion process to parse the entered time duration.

Let's spin up a new module that helps us deal with these conversions. We create a new module in the `lib/utilities/time-utilities.js` file.

First, we need to have a constant that defines all the units we need for the conversion process:

```
export const UNITS = [{
  short: 'w',
  milliseconds: 5 * 8 * 60 * 60 * 1000
}, {
  short: 'd',
  milliseconds: 8 * 60 * 60 * 1000
}, {
  short: 'h',
  milliseconds: 60 * 60 * 1000
}, {
  short: 'm',
  milliseconds: 60 * 1000
}];
```

This is all the units that we need to deal with for now. You can see the milliseconds being calculated at interpretation time. We can also write the milliseconds as constants, but this provides us with some transparency on how we get to these values and we can spear on some comments.

Let's look at our parsing function, which we can use to parse text input into time durations:

```
export function parseDuration(formattedDuration) {
  const pattern = /[\d\.]+\s*[wdhm]/g;
  let timeSpan = 0;
  let result;
  while (result = pattern.exec(formattedDuration)) {
    const chunk = result[0].replace(/\s/g, '');
    let amount = Number(chunk.slice(0, -1));
    let unitShortName = chunk.slice(-1);
    timeSpan += amount * UNITS.find(
      (unit) => unit.short === unitShortName
```

```
    ).milliseconds;
  }
  return +timeSpan || null;
}
```

Let's analyze the preceding code briefly to explain what we do here:

1. First, we define a regular expression that helps us dissect the text representation of a duration. This pattern will extract chunks from the text input that are important to calculate the duration behind the text representation. These chunks always consist of a number, followed by either `w`, `d`, `h`, or `m`. Therefore, the text `10w 3d 2h 30m` will be split into the chunks `10w`, `3d`, `2h`, and `30m`.
2. We initialize a `timeSpan` variable with `0`, so we can add all the milliseconds from discovered chunks together and later return this sum.
3. For each of the previously-extracted chunks, we now extract the number component into a variable called `amount`, and the unit (`w`, `d`, `h`, or `m`) into a variable called `unitShortName`.
4. Now, we can look up the data in the `UNITS` constant for the unit of the chunk that we will process, multiply the amount of milliseconds of the unit by the amount we extract from the chunk, and then add that result to our `timeSpan` variable.

Well this is quite a neat function we built here. It accepts a formatted time duration string and converts it into milliseconds. This is already half of what we need to deal with textual representation of time durations. The second piece is the opposite of what we have with the `parseDuration` function to convert a duration in milliseconds into a formatted duration string:

```
export function formatDuration(timeSpan) {
  return UNITS.reduce((str, unit) => {
    const amount = timeSpan / unit.milliseconds;
    if (amount >= 1) {
      const fullUnits = Math.floor(amount);
      const formatted = `${str} ${fullUnits}${unit.short}`;
      timeSpan -= fullUnits * unit.milliseconds;
      return formatted;
    } else {
      return str;
    }
  }, '').trim();
}
```

Let's also explain briefly what the `formatDuration` function does:

- We use the `Array.prototype.reduce` function to format a string that contains all time units and their amount. We iterate over all available time units in the `UNITS` constant starting with the largest unit for weeks.
- We then divide the `timeSpan` variable, which is in milliseconds, by the milliseconds of the unit which gives us the amount of the given unit.
- If the amount is greater than or equal to 1, we can add the unit with the given amount and unit short name to our formatted string.
- As we could be left with some fractions after the comma in the amount, which we will

need to encode in smaller units, we subtract the floored version of our amount from the `timeSpan` before we return to the `reduce` function again.
- This process is repeated for every unit, where each unit will only provide formatted output if the amount is greater than or equal to 1.

This is all we need to convert back and forth between formatted time duration and time duration represented in milliseconds.

We'll do one more thing before we create the actual component to enter time durations. We will create a simple pipe that basically just wraps our `formatTime` function. For this, we will create a new `lib/pipes/format-duration.js` file:

```
import {Pipe, Inject} from '@angular/core';
import {formatDuration} from '../utilities/time-utilities';

@Pipe({
  name: 'formatDuration'
})
export class FormatDurationPipe {
  transform(value) {
    if (value == null || typeof value !== 'number') {
      return value;
    }

    return formatDuration(value);
  }
}
```

Using the `formatTime` function of our `time-utilities` module, we now have the ability to format durations in millisecond directly from in our templates.

Components to manage efforts

Okay, this is enough time math for the moment. Let's now use the elements that we created to shape some components that will help us gather user input.

In this section, we will create two components to manage efforts:

- `Duration`: The `Duration` component is a simple UI component, enabling user input of time durations using the formatted time strings we dealt with in the previous topics. It uses an `Editor` component to enable user input and makes use of the `FormatTimePipe` pipe as well as the `parseDuration` utility function.
- `Efforts`: The `Efforts` component is just a composition of two `Duration` components that represent the estimated effort and the effective effort spent on a given task. Following a strict rule of composition, this component is important for us so that we don't repeat ourselves and instead compose a larger component.

Let's start with the `Duration` component class, and create a new `lib/ui/duration/duration.js` file:

```
…
import {FormatDurationPipe} from '../../pipes/format-duration';
import {Editor} from '../../ui/editor/editor';
import {parseDuration} from '../../utilities/time-utilities';

@Component({
  selector: 'ngc-duration',
  …
  directives: [Editor],
  pipes: [FormatDurationPipe]
})
export class Duration {
  @Input() duration;
  @Output() durationChange = new EventEmitter();

  onEditSaved(formattedDuration) {
    this.durationChange.next(formattedDuration ?
      parseDuration(formattedDuration) : null);
  }
}
```

There's nothing fancy about this component really because we created the bulk of the logic already and we simply compose a higher component together.

As the `duration` input, we expect a time duration in milliseconds, while the `durationChange` output property will emit events when the user provides some input.

The `onEditSaved` method serves in the binding to the `Editor` component in our component. Whenever the user saves his edits on the `Editor` component, we'll take this input, convert the formatted time duration into milliseconds using the `parseDuration` function, and re-emit the converted value using the `durationChange` output property.

Let's look at the template of our component in the `lib/ui/duration/duration.html` file:

```
<ngc-editor [content]="duration | formatDuration"
            [showControls]="true"
            (editSaved)="onEditSaved($event)"></ngc-editor>
```

Surprised with how simple our template is? Well, this is exactly what we should achieve with higher components once we establish a good foundation of base components. Well-organized composition radically simplifies our code. The only thing that we deal with here is our good old `Editor` component.

We bind the `duration` input property of our `Duration` component to the content input property of the `Editor` component. As we'd like to pass the formatted time duration and not the duration in milliseconds, we use the `FormatDurationPipe` pipe to convert in the binding expression.

If the `Editor` component notifies us about a saved edit, we call the `onEditSaved` method on our `Duration` component, which will parse the entered duration and re-emit the resulting value.

As we initially defined all efforts to consist of an estimated and an effective duration, we would now like to create another component that combines these two durations.

Let's create a new `Efforts` component by starting with a new template on the `lib/efforts/efforts.html` path:

```
<div class="efforts__label">Estimated:</div>
<ngc-duration [duration]="estimated"
              (durationChange)="onEstimatedChange($event)">
</ngc-duration>
<div class="efforts__label">Effective:</div>
<ngc-duration [duration]="effective"
              (durationChange)="onEffectiveChange($event)">
</ngc-duration>
<button class="button button--small"
        (click)="addEffectiveHours(1)">+1h</button>
<button class="button button--small"
        (click)="addEffectiveHours(4)">+4h</button>
<button class="button button--small"
        (click)="addEffectiveHours(8)">+1d</button>
```

First, we add two `Duration` components labelled, where the first one is used to gather input for the estimated time and the later one for effective time.

In addition to this, we provide three small buttons to increase the effective duration by a simple click. In this way, the user can quickly add one or four hours (half a working day) or a complete working day (which we defined as eight hours).

Looking at the `Component` class, there should be no surprises. Let's open the

`lib/efforts/efforts.js` component class file:

```
…
import {Duration} from '../ui/duration/duration';
import {UNITS} from '../utilities/time-utilities';

@Component({
  selector: 'ngc-efforts',
  …
  directives: [Duration]
})
export class Efforts {
  @Input() estimated;
  @Input() effective;
  @Output() effortsChange = new EventEmitter();

  onEstimatedChange(estimated) {
    this.effortsChange.next({
      estimated,
      effective: this.effective
    });
  }

  onEffectiveChange(effective) {
    this.effortsChange.next({
      effective,
      estimated: this.estimated
    });
  }

  addEffectiveHours(hours) {
    this.effortsChange.next({
      effective: (this.effective || 0) +
        hours * UNITS.find((unit) => unit.short === 'h'),
      estimated: this.estimated
    });
  }
}
```

The component provides two separate inputs for estimated and effective time duration in milliseconds. If you take a look at the component template again, these input properties are directly bound to the input properties of the `Duration` components.

The `onEstimatedChange` and `onEffectiveChange` methods are used to create bindings to the `durationChange` output properties of the `Duration` components. All we do here is emit an aggregated data object that contains the effective and estimated time in milliseconds using the `effortsChange` output property.

In the `addEffectiveHours` method, we simply emit an `effortsChange` event and update the effective property by the calculated amount of milliseconds. We use our `UNITS` constant from the `time-utilities` module in order to get the amount of milliseconds for an hour.

This is all that we need in order to provide a user input to manage efforts on our tasks. To

complete this topic, we will add our newly-created `Efforts` component to the `ProjectTaskDetail` component in order to manage efforts on tasks.

Let's first look at the code changes in the `Component` class located in `lib/project/project-task-detail/project-task-detail.js`:

```
…
import {Efforts} from '../../efforts/efforts';

@Component({
  selector: 'ngc-project-task-details',
  …
  directives: [Editor, Efforts]
})
export class ProjectTaskDetails {
  …
  onEffortsChange(efforts) {
    if (!efforts.estimated && !efforts.effective) {
      this.task.efforts = null;
    } else {
      this.task.efforts = efforts;
    }
    this.project.document.persist();
  }
  …
}
```

Besides providing the `Efforts` component to the `directives` list of our `ProjectTaskDetail` component, we added a new `onEffortsChange` method that deals with the output provided by the `Efforts` component.

If both estimated and effective effort isn't set, or set to `0`, we'll set the task efforts to `null`. Otherwise, we use the output data of the `Efforts` component and assign it as our new task efforts.

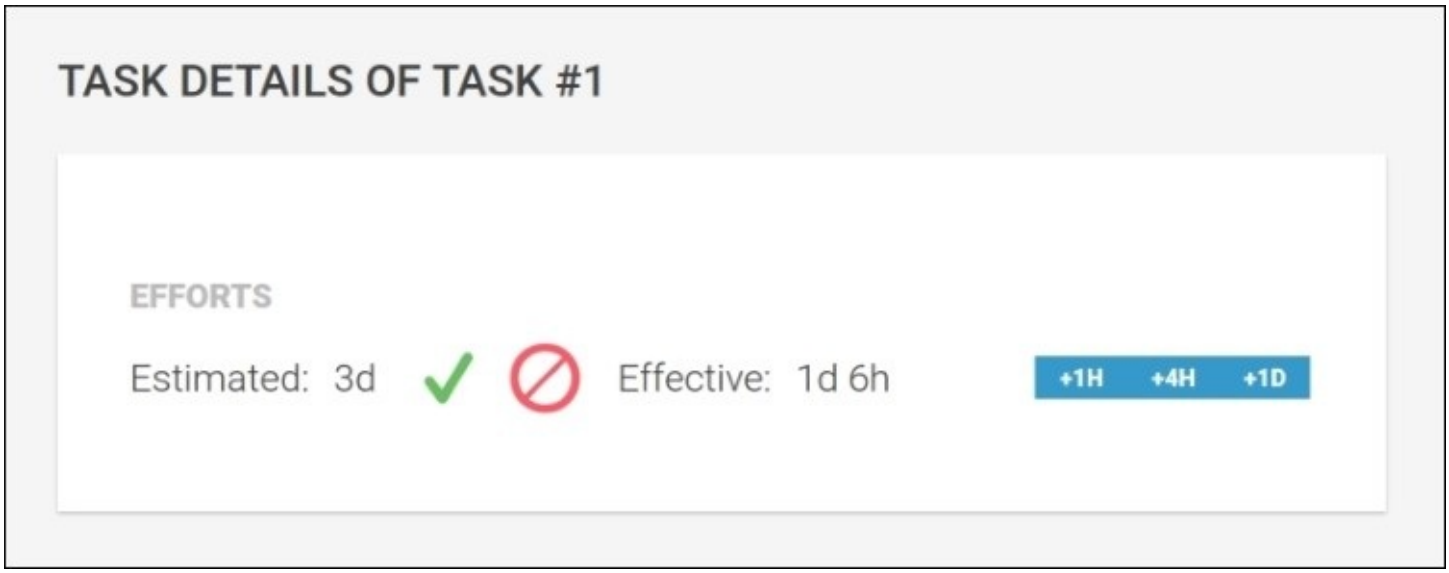
After changing the task efforts, we persist the `LiveDocument` of the project in the same way that we do for the title and the description updates already.

Let's check out the changes in the template of our component located in `lib/project/project-task-detail/project-task-detail.html`:

```
…
<div class="task-details__content">
  …
  <div class="task-details__label">Efforts</div>
  <ngc-efforts [estimated]="task?.efforts?.estimated"
               [effective]="task?.efforts?.effective"
               (effortsChange)="onEffortsChange($event)">
  </ngc-efforts>
</div>
```

We are binding the estimated and effective input properties of the `Efforts` component to the

task data in the `ProjectTaskDetail` component. For the `effortsChange` output property we're using an expression that is invoking our `onEffortsChange` method that we've just created:



Our new Efforts component that consists of two duration input components

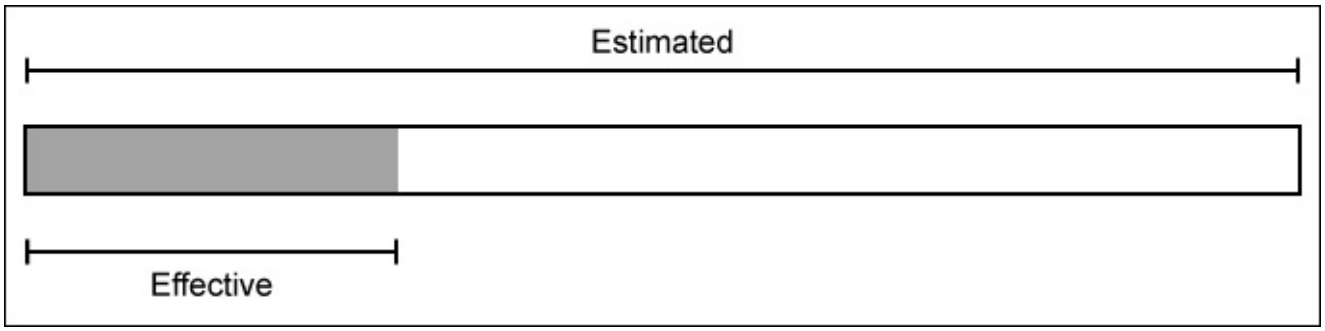
The visual efforts timeline

Although the components that we created so far to manage efforts provide a good way to edit and display effort and time durations, we can still improve this with some visual indication.

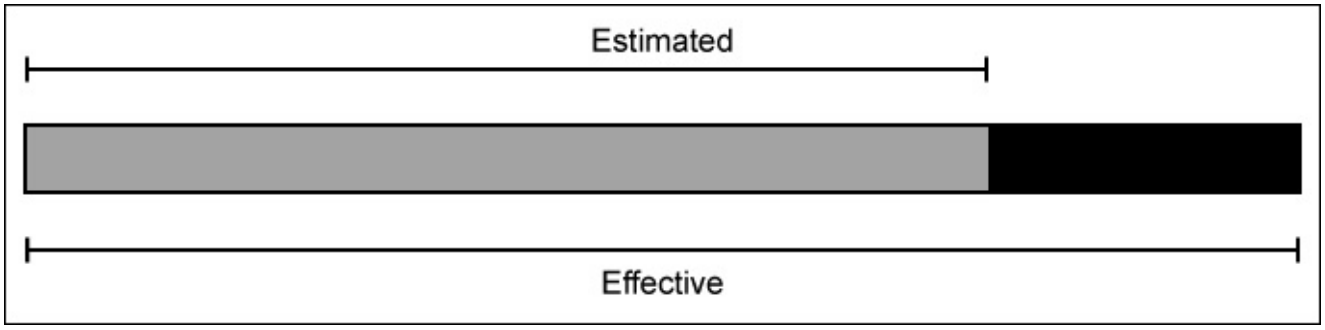
In this section, we will create a visual efforts timeline using SVG. This timeline should display the following information:

- The total estimated duration as a gray background bar
- The total effective duration as a green bar that overlays on the total estimated duration bar
- A yellow bar that shows any overtime (if the effective duration is greater than the estimated duration)

The following two figures illustrate the different visual states of our efforts timeline component:



The visual state if the estimated duration is greater than the effective duration



The visual state if the effective duration exceeds the estimated duration (the overtime is displayed as a black bar)

Let's start fleshing out our component by creating a new `EffortsTimeline` Component class on the `lib/efforts/efforts-timeline/efforts-timeline.js` path:

```
…
@Component({
  selector: 'ngc-efforts-timeline',
  …
})
export class EffortsTimeline {
  @Input() estimated;
  @Input() effective;
  @Input() height;

  ngOnChanges(changes) {
    this.done = 0;
    this.overtime = 0;

    if (!this.estimated && this.effective ||
        (this.estimated && this.estimated === this.effective)) {
      // If there's only effective time or if the estimated time
      // is equal to the effective time we are 100% done
      this.done = 100;
    } else if (this.estimated < this.effective) {
      // If we have more effective time than estimated we need to
      // calculate overtime and done in percentage
      this.done = this.estimated / this.effective * 100;
      this.overtime = 100 - this.done;
    } else {
      // The regular case where we have less effective time than
      // estimated
      this.done = this.effective / this.estimated * 100;
    }
  }
}
```

Our component has three input properties:

- `estimated`: This is the estimated time duration in milliseconds
- `effective`: This is the effective time duration in milliseconds
- `height`: This is the desired height of the efforts timeline in pixels

In the `OnChanges` lifecycle hook, we set two component member fields, which are based on the estimated and effective time:

- `done`: This contains the width of the green bar in percentage that displays the effective duration without overtime that exceeds the estimated duration
- `overtime`: This contains the width of the yellow bar in percentage that displays any overtime, which is any time duration that exceeds the estimated duration

Let's look at the template of the `EffortsTimeline` component and see how we can now use the `done` and `overtime` member fields to draw our timeline.

We will create a new `lib/efforts/efforts-timeline/efforts-timeline.html` file:

```
<svg width="100%" [attr.height]="height">
  <rect [attr.height]="height"
```

```
        x="0" y="0" width="100%"
        class="efforts-timeline__remaining"></rect>
  <rect *ngIf="done" x="0" y="0"
        [attr.width]="done + '%'" [attr.height]="height"
        class="efforts-timeline__done"></rect>
  <rect *ngIf="overtime" [attr.x]="done + '%'" y="0"
        [attr.width]="overtime + '%'" [attr.height]="height"
        class="efforts-timeline__overtime"></rect>
</svg>
```

Our template is SVG-based, and it contains three rectangles for each of the bars that we want to display. The background bar that will be visible if there is remaining effort will always be displayed.

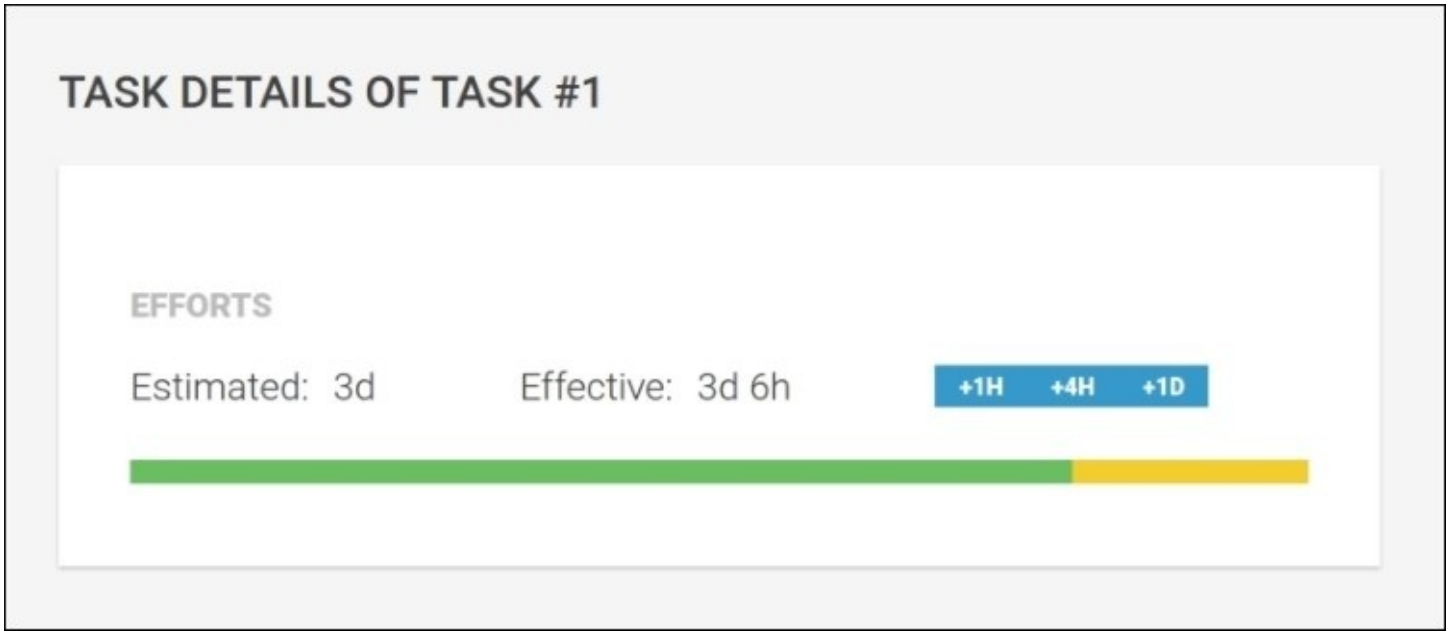
Above the remaining bar, we conditionally display the done and the overtime bar using the calculated widths from our component class.

Now, we can go ahead and include the `EffortsTimeline` class in our `Efforts` component. This way our users will have visual feedback when they edit the estimated or effective duration, and it provides them a sense of overview.

Let's look into the template of the `Efforts` component to see how we integrate the timeline:

```
…
<ngc-efforts-timeline height="10"
                      [estimated]="estimated"
                      [effective]="effective">
</ngc-efforts-timeline>
```

As we have the estimated and effective duration times readily available in our `Efforts` component, we can simply create a binding to the `EffortsTimeline` component input properties:



The Efforts component displaying our newly-created efforts timeline component (the overtime of six hours is visualized with the yellow bar)

Recapitulating on efforts management

In this section, we'll create components that allow users to manage efforts easily and add a simple but powerful time tracking to our tasks. We've done the following to achieve this:

- We implemented some utility functions to deal with the time math in order to convert time durations in milliseconds into formatted time durations and vice versa
- We created a pipe to format time durations in milliseconds using our utility functions
- We created a `Duration` UI component, which wraps an `Editor` component and uses our time utilities to provide a no-UI kind of input element to enter durations
- We created an `Efforts` component that acts as a composition of two `Duration` components for estimated and effective time and provides additional buttons to add effective spent time quickly
- We integrated the `Efforts` component into the `ProjectTaskDetail` component in order to manage efforts on tasks
- We created a visual `EffortsTimeline` component using SVG, which displays the overall progress on a task

Setting milestones

Tracking time is important. I don't know how you feel about time, but I really suck at organizing my time. Although a lot of people ask me how I manage to do so many things, I believe I'm actually very bad at managing how I get these things done. If I were a better organizer, I could get things done with much less energy involved.

One thing that always helps me organize myself is to break things down into smaller work packages. Users that organize themselves with our task management application can already do this by creating tasks in projects. While a project is the overall goal, we can create smaller tasks to achieve this goal. However, sometimes we tend to lose sight of the overall goal when we're only focused on tasks.

Milestones are a perfect glue between projects and tasks. They make sure that we bundle tasks together into larger packages. This will help us a lot in organizing our tasks, and we can look at milestones of the project to see the overall project health. However, we can still focus on tasks when we work in the context of a milestone.

In this section, we will create the necessary components in order to add basic milestone functionality to our application.

To implement milestone functionality in our application, we will stick to the following design decisions:

- Milestones should be stored on the project level, and tasks can contain an optional reference to a project milestone.
- To keep things simple, the only interaction point with milestones should be on task level. Therefore, creation of milestones will be done on task level, although the created milestones will be stored on project level.
- Milestones currently only consist of a name. There are a lot more to milestones that we can potentially build into our system, such as deadlines, dependencies, and other nice things. However, we will stick to the bare minimum, which is a milestone name.

Creating an autocomplete component

In order to keep the management of milestones simple, we will create a new user interface component to deal with the design concerns that we listed. Our new autocomplete component will not only display possible values to select from, but it will also allow us to create new items. We can then simply use this component on our `ProjectTaskDetail` component in order to manage milestones.

Let's look at the `Component` class of our new autocomplete component that we will create in the `lib/ui/auto-complete/auto-complete.js` file:

```
…
import {Editor} from '../editor/editor';

@Component({
  selector: 'ngc-auto-complete',
  …
  directives: [Editor]
})
export class AutoComplete {
  @Input() items;
  @Input() selectedItem;
  @Output() selectedItemChange = new EventEmitter();
  @Output() itemCreated = new EventEmitter();
  …
}
```

Once again, our `Editor` component can be reused to create this higher component. We're lucky that we created such a nice component, as this saved us a lot of time throughout this project.

Let's look at the input and output properties of the `AutoComplete` component in more detail:

- `items`: This is where we expect an array of strings. This will be the list of items a user can choose from when typing into the editor.
- `selectedItem`: This is when we make the selected item an input property to actually make this component pure, and we can rely on the parent component to set this property right.
- `selectedItemChange`: This output property will emit an event if the selected item was changed. As we create a pure component here, we somehow need to propagate the event of an item that was selected in the autocomplete list.
- `itemCreated`: This output property will emit an event if a new item was added to the autocomplete list. Updating the list of items and changing the component `items` input property will still be the responsibility of the parent component.

Let's add more code to our component. We use an `Editor` component as main input source. While our users will type into the editor, we filter the available items using the text input of the editor. Let's create a `filterItems` for this purpose:

```
filterItems(filter) {
```

```
  this.filter = filter || '';
  this.filteredItems = this.items
    .filter(
      (item) => item
        .toLowerCase()
        .indexOf(this.filter.toLowerCase().trim()) !== -1)
    .slice(0, 10);
  this.exactMatch = this.items.includes(this.filter);
}
```

The `filterItems` method has a single parameter, which is the text that we want to use in order to search for relevant items in our list.

Let's look at the content of the method in more detail:

- For later use in our template, we will set aside the filter query that was used the last time this method was called
- In the `filteredItems` member variable, we will store a filtered version of the item list by searching for text occurrences of the filter string
- As a last step, we also store the information if the search query resulted in an exact match of an item in our list

Now, we need to make sure that if the `items` or `selectedItem` input properties change, we also execute our filter method again. For this, we simply implement the `ngOnChanges` lifecycle hook:

```
ngOnChanges(changes) {
  if (this.items && this.selectedItem) {
    this.filterItems(this.selectedItem);
  }
}
```

Let's now see how we deal with the events provided by the `Editor` component:

```
onEditModeChange(editMode) {
  if (editMode) {
    this.showCallout = true;
    this.previousSelectedItem = this.selectedItem;
  } else {
    this.showCallout = false;
  }
}
```

If the editor changes to edit mode, we want to save the previously selected item. We'll need this if the user decides to cancel his edits and switch back to the previous item. Of course, this is also the point where we need to display the autocomplete list to the user.

On the other hand, if the edit mode is switched back to read mode, we want to hide the autocomplete list again:

```
onEditableInput(content) {
  this.filterItems(content);
```

```
}
```

The `editableInput` event is triggered by our editor on every editor input change. The event provides us with the text content that was entered by the user. If such an event occurs, we need to execute our filter function again with the updated filter query:

```
onEditSaved(content) {
  if (content === '') {
    this.selectedItemChange.next(null);
  } else if (content !== this.selectedItem &&
             !this.items.includes(content)) {
    this.itemCreated.next(content);
  }
}
```

When the `editSaved` event is triggered by our editor, we need to decide whether we should do either of the following:

- Emit an event using the `selectedItemChange` output property if the saved content is an empty string to signal the removal of a selected item to the parent component
- Emit an event using the `itemCreated` output property if valid content is given and our list does not include an item with that name to signal an item creation:

```
onEditCanceled() {
  this.selectedItemChange.next(this.previousSelectedItem);
}
```

On the `editCanceled` event of the `Editor` component, we want to switch back to the previous selected item. For this, we can simply emit an event using the `selectedItemChange` output property and the `previousSelectedItem` member that we put aside after the editor was switched into edit mode.

These are all the binding functions that we will use to wire up our editor and in order to attach the autocomplete functionality to it.

There are two more rather simple methods that we will create before we take a look at the template of our autocomplete component:

```
selectItem(item) {
  this.selectedItemChange.next(item);
}

createItem(item) {
  this.itemCreated.next(item);
}
```

We will use these two for the click actions in the autocomplete callout from our template. Let's take a look at the template so that you can see all the code that we just created in action:

```
<ngc-editor [content]="selectedItem"
            [showControls]="true"
```

```
            (editModeChange)="onEditModeChange($event)"
            (editableInput)="onEditableInput($event)"
            (editSaved)="onEditSaved($event)"
            (editCanceled)="onEditCanceled($event)"></ngc-editor>
```

First, the `Editor` component is placed and all necessary bindings to the handler methods that we created in our `Component` class are attached.

Now, we will create the autocomplete list that will be displayed as a callout to the user right next to the editor input area:

```
<ul *ngIf="showCallout" class="auto-complete__callout">
  <li *ngFor="let item of filteredItems"
      (click)="selectItem(item)"
      class="auto-complete__item"
      [class.auto-complete__item--selected]="item === selectedItem">{{item}}</li>
  <li *ngIf="filter && !exactMatch"
      (click)="createItem(filter)"
      class="auto-complete__item auto-complete__item--create">Create "{{filter}}"
</li>
</ul>
```

We rely on the `showCallout` member set by the `onEditModeChange` method of our `Component` class to signal if we should display the autocomplete list or not.

We then iterate over all filtered items using the `NgFor` directive and render the text content of each item. If one of the items gets clicked on, we will call our `selectItem` method with the concerned item as the parameter value.

As the last list element, after the repeated list items, we conditionally display an additional list element in order to create a nonexisting milestone. We only display this button if there's a valid filter already and if there's no exact match of the filter to an existing milestone:

Our milestone component plays nicely together with the editor component using a clean composition

Now that we are all done with our autocomplete component, the only thing left to do in order to manage project milestones is to make use of it in the `ProjectTaskDetails` component.

Let's open the `Component` class located in `lib/project/project-task-details/project-task-details.js` and apply the necessary modifications:

```
…
import {AutoComplete} from '../../ui/auto-complete/auto-complete';

@Component({
  selector: 'ngc-project-task-details',
  …
  directives: […, AutoComplete]
})
export class ProjectTaskDetails {
  constructor(@Inject(forwardRef(() => Project)) project, {
    …
    this.projectChangeSubscription =
this.project.document.change.subscribe((data) => {
      …
      this.projectMilestones = data.milestones || [];
    });
  }
  …
  onMilestoneSelected(milestone) {
    this.task.milestone = milestone;
    this.project.document.persist();
  }

  onMilestoneCreated(milestone) {
    this.project.document.data.milestones =
this.project.document.data.milestones || [];
    this.project.document.data.milestones.push(milestone);
    this.task.milestone = milestone;
    this.project.document.persist();
  }
  …
}
```

In the subscription to project changes, we now also extract any preexisting project milestones and store them in a `projectMilestones` member variable. This makes it easier to reference in the template.

The `onMilestoneSelected` method will be bound to the `selectItemChange` output property of the `AutoComplete` component. We use the emitted value of the `AutoComplete` component to set our tasks milestone and persist the `LiveDocument` project using its `persist` method.

The `onMilestoneCreated` method will be bound to the `itemCreated` output property of the `AutoComplete` component. On such an event, we add the created milestone to the projects milestone list as well as assign the current task to the created milestone. After updating the `LiveDocument` data, we use the `persist` method to save all changes.

Let's look into `lib/project/project-task-details/project-task-details.html` to see the necessary changes in our template:

```
…
<div class="task-details__content">
  …
  <ngc-auto-complete [items]="projectMilestones"
                     [selectedItem]="task?.milestone"
                (selectedItemChange)="onMilestoneSelected($event)"
                (itemCreated)="onMilestoneCreated($event)">
  </ngc-auto-complete>
</div>
```

Besides the output property bindings that you're already aware of, we also create two input bindings for the `items` and `selectedItem` input properties of the `AutoComplete` component.

This is already it. We created a new UI component that provides autocompletion and used that component to implement milestone management on our tasks.

Isn't it nice how easy it suddenly seems to implement new functionality when using components with proper encapsulation? The great thing about component-oriented development is that your development time for new functionality decreased with the amount of reusable components that you already created.

Summary

In this chapter, we implemented some components that help our users keep track of time. They can now log efforts on tasks and manage milestones on projects. We created a new task detail view that can be accessed using a navigation link on our task list.

Once more, we experienced the power of composition using components, and reusing existing components, we were able to easily implement higher components that provide more complex functionality.

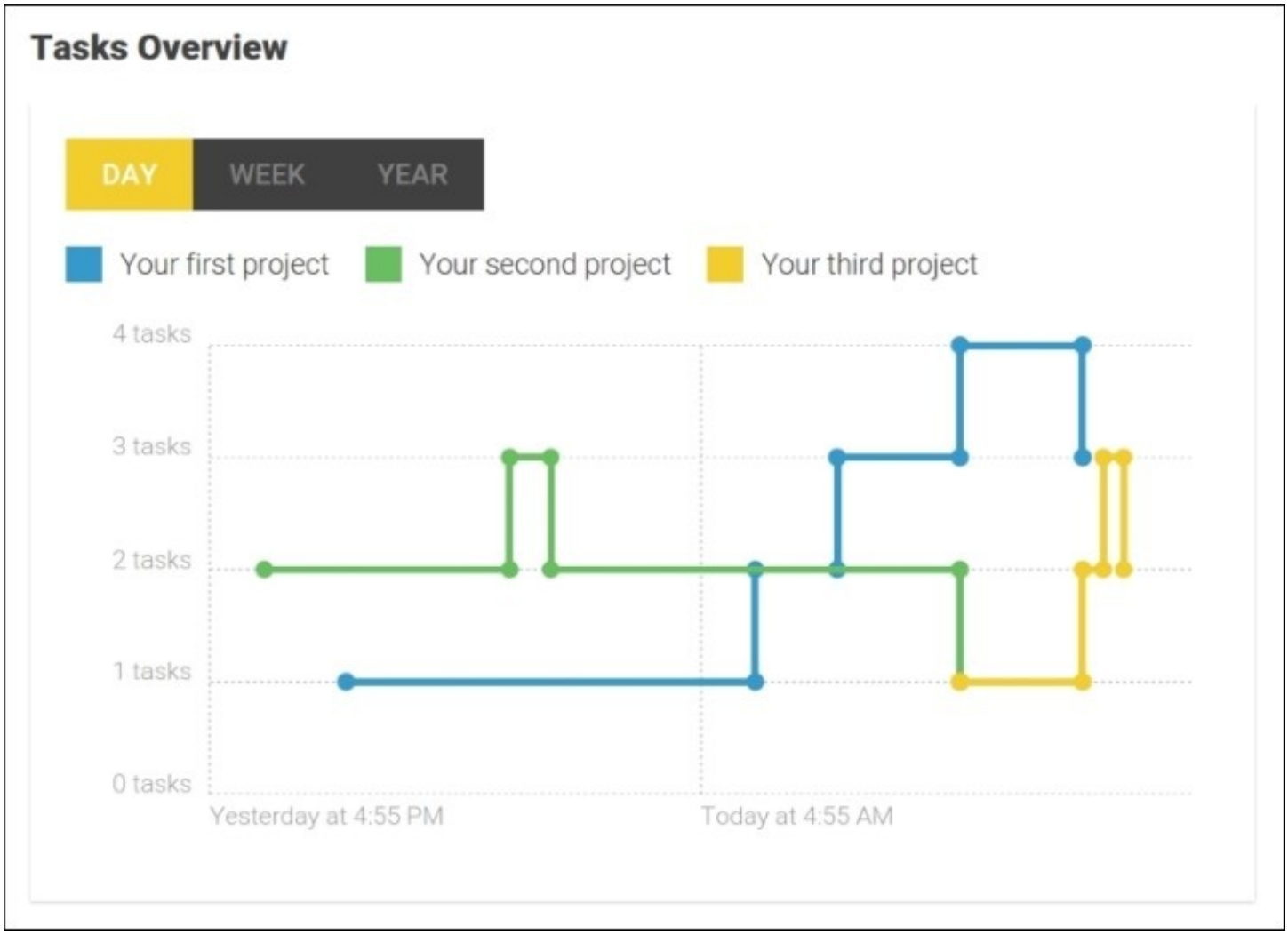
In the next chapter, we will look at how to use the charting library Chartist and create some wrapper components that allow us to build reusable charts. We will build a dashboard for our task management system, where we will see our chart components in action.

Chapter 9. Spaceship Dashboard

When I was a child, I loved to play spaceship pilot. I piled up old carton boxes and decorated the interior to look like a spaceship cockpit. With a marker, I drew a spaceship dashboard on the inside of the boxes, and I remember playing in there for hours.

The thing that's special about the design of cockpits and spaceship dashboards is that they need to provide an overview and control over the whole spaceship on very limited space. I think the same also applies to application dashboards. A dashboard should provide the user with an overview and a sense for the overall status of what's going on.

In this chapter, we will create such a dashboard for our task management application. We will make use of the open source charting library Chartist to create good looking responsive charts and provide an overview over open tasks and project status:



A preview of the tasks chart that we will build during the course of this chapter

On a higher level, we will create the following components in this chapter:

- **Project summary**: This is the project summary that will provide a quick insight into the overall project status. By aggregating all efforts of containing tasks, we can provide a nice overall efforts status, for which we have created the components in the previous chapter.
- **Project activity chart**: Without any labels or scales, this bar chart will just give a quick sense for the activity on projects in the last 24 hours.
- **Project tasks chart**: This chart provides an overview of the task progress on projects. Using a line chart, we will display the count of open tasks over a certain time period. Using our Toggle component that we created in Chapter 2, *Ready, Set, Go!*, of this book, we'll provide an easy way for the user to switch the displayed timeframe on the chart.

Introduction to Chartist

We will create some components in this chapter that will render charts, and we should look for some help in rendering them. Of course, we can follow a similar approach as we did in Chapter 6, *Keeping Up with Activities*, when we drew our activity timeline. However, when it comes to more complex data visualization, it's better to rely on a library to do the heavy lifting.

It shouldn't be a surprise that we'll use Chartist to fill this gap because I've spent almost two years writing it. As the author of Chartist, I feel very lucky that we've found a perfect spot in this book to make use of it.

I'd like to take the opportunity to introduce you to Chartist briefly before we dive into the implementation of the components for our dashboard.

The claim of Chartist is simple responsive charts, and this is luckily still the case after three years of existence. I can tell you that probably the hardest job of maintaining this library was to protect it from feature bloating. There are so many great movements, technologies, and ideas in the open source community and to resist and always stay focused on the initial claim wasn't easy.

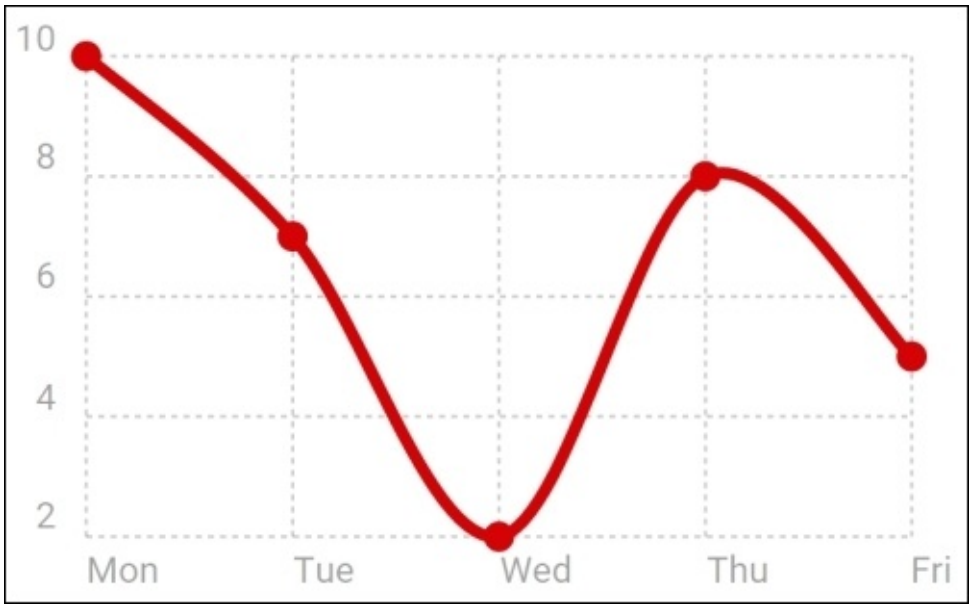
Let me show you a very basic example of how you can create a simple line chart once you've included the Chartist scripts on your website:

```
const chart = new Chartist.Line('#chart', {
labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
series: [
    [10, 7, 2, 8, 5]
  ]
});
```

The corresponding HTML markup that is required for this example looks as simple as the following:

```
<body>
<div id="chart" class="ct-golden-section"></div>
</body>
```

The following figure shows you the resulting chart that is rendered by Chartist:



A simple line chart rendered with Chartist

I believe that by saying that we've been sticking to our claim to stay simple, we've not promised too much.

Let's look at the second core concern of Chartist, which is to be perfectly responsive. Well, let's start with one of my most appreciated principles in web development, which is the separation of concerns in the frontend. Chartist tries to stick to this separation wherever possible, which means that it uses CSS for its appearance, SVG for the basic graphical structure, and JavaScript for any behavior. Simply by following this principle, we've already enabled a lot of responsiveness. We can use CSS media queries to apply different styles to our charts on different media.

While CSS is great for visual styles, there are plenty of elements in the process of rendering charts, which can't be controlled simply by styling. After all, this is the reason why we use a JavaScript library to render charts.

So, how can we control how Chartist renders our charts on different media if we haven't got control over this in CSS? Well, Chartist provides something called responsive configuration overrides. Using the browsers `matchMedia` API, Chartist is able to provide a configuration mechanism that allows you to specify options that you want overridden on certain media.

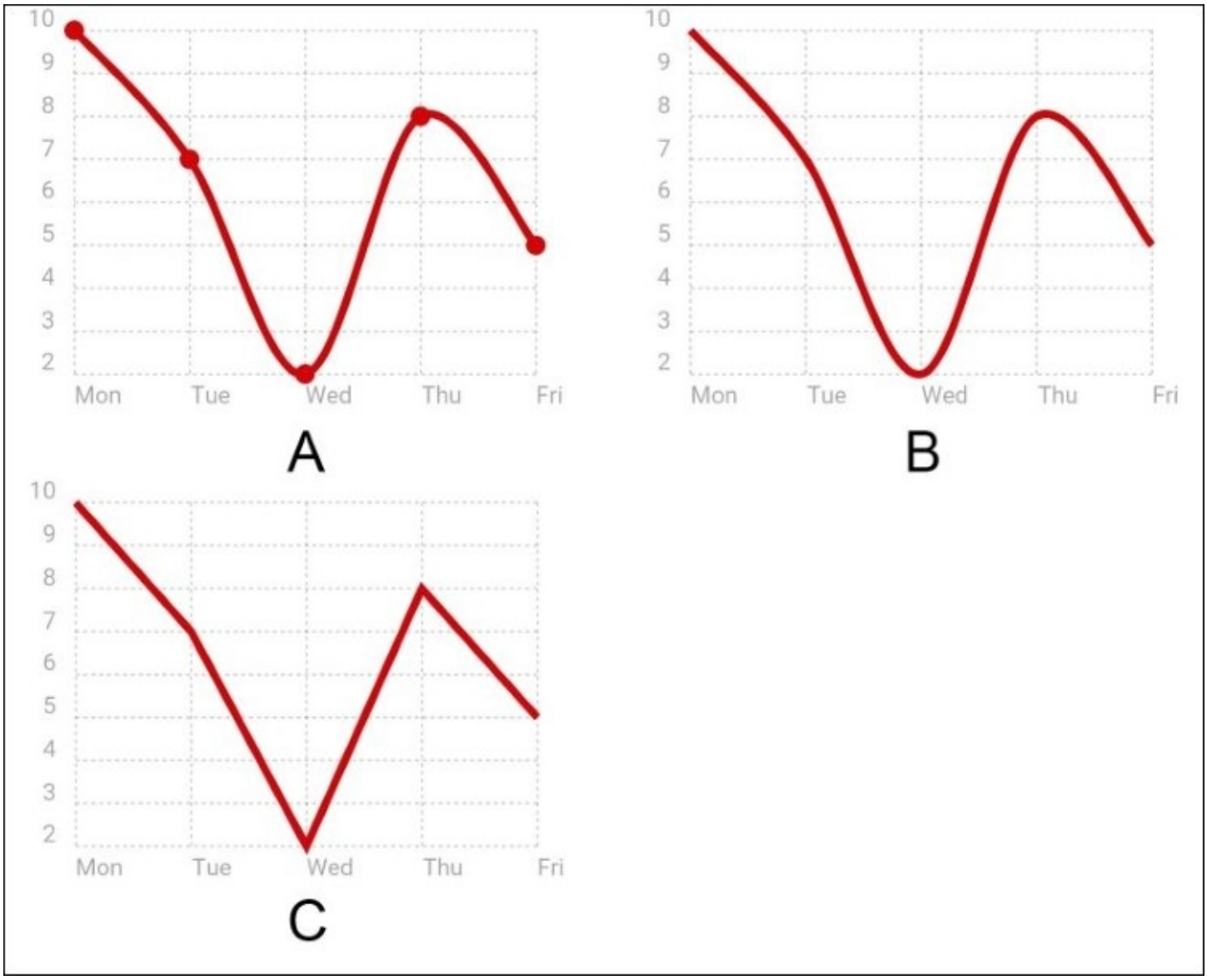
Let's look into a simple example of how we can easily implement responsive behavior using a mobile-first approach:

```
const chart = new Chartist.Line('#chart', {
labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
series: [
    [10, 7, 2, 8, 5]
```

```
  ]
}, {
showPoint: true,
showLine: true
}, [
  ['screen and (min-width: 400px)', {
showPoint: false
  }],
  ['screen and (min-width: 800px)', {
lineSmooth: false
  }]
]);
```

Here, the second parameter to the `Chartist.Line` constructor sets the initial options; we can provide overriding options annotated with media queries in an array as the third parameter of the constructor. In this example, we'll override the `showPoint` option for any media larger than 400 px in width. Media larger than 800 px in width will receive both the `showPoint` override as well as the `lineSmooth` override.

Not only can we specify real media queries to trigger setting changes, but we can also use an overriding mechanism that is very similar to CSS. This way, we can implement various approaches, such as ranged or exclusive media queries, mobile-first, or desktop-first. This responsive options mechanism can be used for all options available in Chartist.



Displaying the previous chart on three different media left to right, starting from a media with less than 400 px (A), less than 800 px (B), and more than 800 px (C).

As you can see, implementing complex responsive behavior is a breeze with Chartist. Although our task management application was never meant to be a responsive web application, we can still benefit from this feature in order to optimize our content.

If I've tickled your fantasy with Chartist, I recommend that you check out the project's website at http://gionkunz.github.io/chartist-js. On the website, you can also visit the live example page at http://gionkunz.github.io/chartist-js/examples.html, where you can hack some charts directly in the browser.

Projects dashboard

In this chapter, we'll create a projects dashboard, which will consist of the following components:

- **Tasks chart**: This is where we'll provide a visual overview on open tasks over time. All projects will be represented in a line chart that displays the progress of open tasks. We'll also provide some user interaction so that the user can choose between different timeframes.
- **Activity chart**: This component visualizes activities in a bar chart over a timeframe of 24 hours. This will help our users quickly identify overall and peak project activities.
- **Project summary**: This is where we'll display a summary of each project where we outline the most important facts. Our project summary component will also include an activity chart component that visualizes project activity.
- **Projects dashboard**: This component is just a composition of the previous two components. This is our main component in the dashboard. It represents our dashboard page and is directly exposed to the router.

Creating the projects dashboard component

First, we'll create our main dashboard component. The `ProjectsDashboard` component has only two responsibilities:

- Obtaining project data, which is used to create the dashboard
- Composing the main dashboard layout by including our dashboard subcomponents

Let's jump right in and create a new component class on the path, `lib/projects-dashboard/projects-dashboard.js`:

```
import {Component, ViewEncapsulation, Inject} from '@angular/core';
import template from './projects-dashboard.html!text';
import {ProjectService} from '../project/project-service/project-service';

@Component({
selector: 'ngc-projects-dashboard',
host: {class: 'projects-dashboard'},
template,
encapsulation: ViewEncapsulation.None
})
export class ProjectsDashboard {
constructor(@Inject(ProjectService) projectService) {
this.projects = projectService.change;
  }
}
```

In our `dashboard` component, we'll use the change observable of `ProjectService` directly. This is different to the usual way that we deal with observables. Usually, we'd subscribe to the observable in our component and then update our component whenever data streams through. However, in our projects dashboard, we're directly storing the change observable of `ProjectService` on our component.

Now, we can use one of Angular's async core pipes in order to subscribe to the observable directly in our view.

Exposing observables directly into the view and using the `async` pipe to subscribe to the observable comes with a main advantage.

We don't need to deal with subscribing and unsubscribing in our component, as the `async` pipe will do that for us directly in the view.

When a new value is emitted within the observable, the `async` pipe will cause the underlying binding to be updated. Also, if the view gets destroyed because of any reason, the `async` pipe will automatically unsubscribe from the observable.

Tip

By chaining RxJS operators together, we can bring an observable stream into the required

shape without performing any subscription. Using the `async` pipe, we can then leave it up to the view to subscribe and unsubscribe from the transformed observable stream. This encourages us to write pure and stateless components, and when used correctly, this is a great practice.

Let's take a look at the view of our component created in the `projects-dashboard.html` file in the same directory as the `Component` class:

```
<div class="projects-dashboard__l-header">
<h2 class="projects-dashboard__title">Dashboard</h2>
</div>
<div class="projects-dashboard__l-main">
<h3 class="projects-dashboard__sub-title">Projects</h3>
<ul class="projects-dashboard__list">
<li *ngFor="let project of projects | async">
<div>{{project.title}}</div>
<div>{{project.description}}</div>
</li>
</ul>
</div>
```

You can see from the template that we use the `async` pipe to subscribe to the projects observable of our `Component` class. The `async` pipe will initially return `null`, but on any change in the observable, this will return the resolved value from the subscription. This means that we don't need to worry about subscribing to our project list observable. We can simply make use of the `async` pipe to subscribe and resolve directly in our view.

For the moment, we only displayed the project title and description, but in the next section, we will create a new project summary component that will deal with some more complex rendering.

Project summary component

In this section, we'll create a `project-summary` component that will provide some overview information for projects. Besides the title and description, this will also include an overview over the total efforts on the project tasks.

Let's first build the component and make the necessary preparations so that we can display the total effort of the underlying tasks of the project.

We'll start with the Component class on the `lib/projects-dashboard/project-summary/project-summary.js` path:

```
...
import{FormatEffortsPipe} from '../../pipes/format-efforts';
import{EffortsTimeline} from '../../efforts/efforts-timeline/efforts-
timeline';
import template from './project-summary.html!text';

@Component({
selector: 'ngc-project-summary',
host: { class: 'project-summary' },
template,
directives: [EffortsTimeline],
pipes: [FormatEffortsPipe],
encapsulation: ViewEncapsulation.None
})
export class ProjectSummary {
@Input() project;

ngOnChanges(changes) {
if (this.project) {
this.totalEfforts = this.project.tasks.reduce(
        (totalEfforts, task) => {
if (task.efforts) {
totalEfforts.estimated += task.efforts.estimated || 0;
totalEfforts.effective += task.efforts.effective || 0;
          }

returntotalEfforts;
        }, {
estimated: 0,
effective: 0
        });
       }
  }
}
```

As you've probably already guessed, we reused the `EffortsTimeline` component that we created in the previous chapter. As our project summary will also include an efforts timeline, based on the same semantics as our total effort, there's no need to create a new component for this.

What we need to do, though, is to accumulate all task efforts into an overall effort. Using the `Array.prototype.reduce` function, we can accumulate all task efforts relatively easy.

The resulting object from the reduce call needs to keep up with the format that is expected of an `efforts` object. As an initial value, we'll provide an `efforts` object with an estimated and effective time of zero. Then, the reduce callback will add any task effort values that are found in the project.

Let's look into the template to see how we're going to use this total efforts data to display our `EffortsTimeline` component:

```
<div class="project-summary__title">{{project?.title}}</div>
<div class="project-summary__description">
{{project?.description}}
</div>
<div class="project-summary__label">Total Efforts</div>
<ngc-efforts-timeline [estimated]="totalEfforts?.estimated"
                      [effective]="totalEfforts?.effective"
height="10"></ngc-efforts-timeline>
<p>{{totalEfforts | formatEfforts}}</p>
```

After displaying the title and description of the project, we included the `EffortsTimeline` component, which we bind to the `totalEfforts` member that we just constructed. This timeline will now display the total aggregated amount of efforts logged on the tasks.

In addition to the timeline, we also rendered a formatted efforts text, such as the one that we already rendered in the `Efforts` component of the previous chapter. For this, we used the `FormatEffortsPipe` pipe.

Now, what's still left to do is to integrate our `ProjectSummary` component into the `ProjectsDashboard` component.

Let's look at the template modification in the `projects-dashboard.html` component template:

```
...
<li *ngFor="let project of projects | async">
<ngc-project-summary
    [project]="project"
    [routerLink]="['/projects', project._id]">
</ngc-project-summary>
</li>
...
```

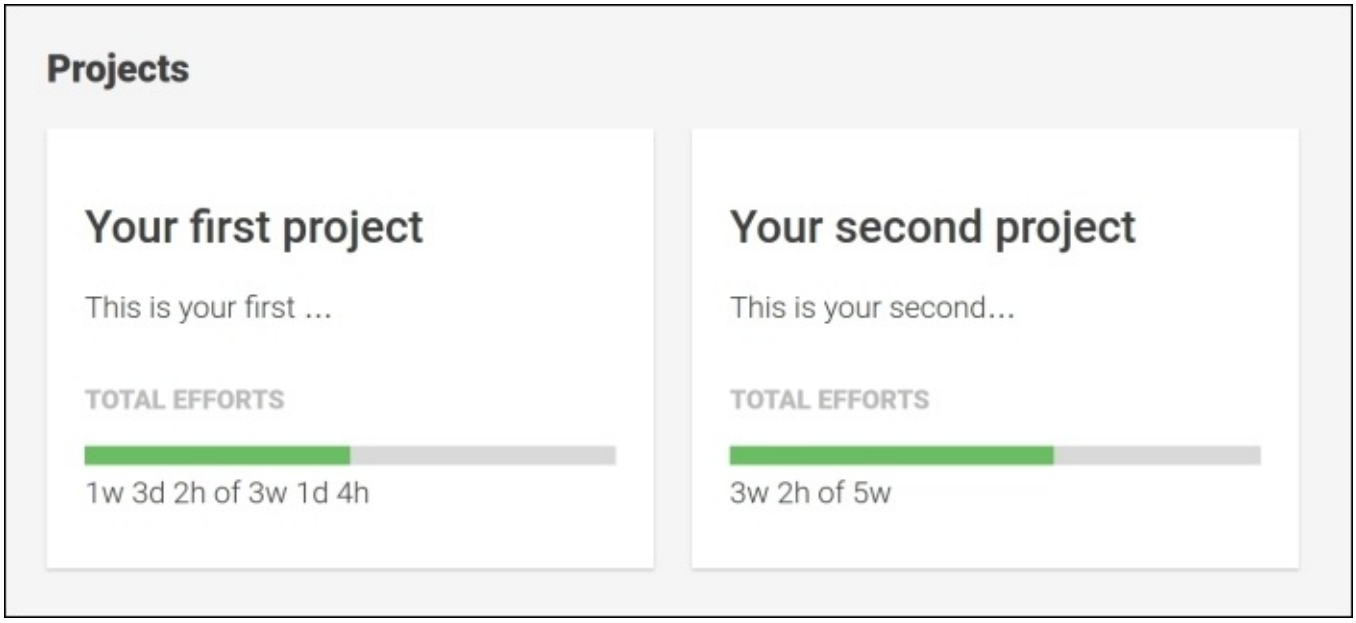
You can see that we bind the `project` local view variable, which was created by the `NgFor` directive in conjunction with the `async` pipe, to the `project` input property of our `ProjectSummary` component.

We also used the `RouterLink` directive to establish the navigation onto the `ProjectDetails` view if the user clicks on one of the summary tiles.

The modifications in the `ProjectsDashboard` component class are negligible:

```
...
import {ROUTER_DIRECTIVES} from '@angular/router';
import{ProjectSummary} from './project-summary/project-summary';
...
@Component({
selector: 'ngc-projects-dashboard',
directives: [ProjectSummary, ROUTER_DIRECTIVES],
  ...
})
export class ProjectsDashboard {
  ...
}
```

The only modification that we applied to the `Component` class was to add the `ProjectSummary` component and the `ROUTER_DIRECTIVES` constant to the directives list of the component. The `ROUTER_DIRECTIVES` constant includes the `RouterOutlet` and `RouterLink` directives, and we use the latter in our template.



A projects dashboard displaying two project summary components with the aggregated total effort

Okay, so far so good. We created two new components and reused our `EffortsTimeline` component to create an aggregated view on the tasks efforts. In the next section, we will enrich our `ProjectSummary` component with a nice Chartist chart.

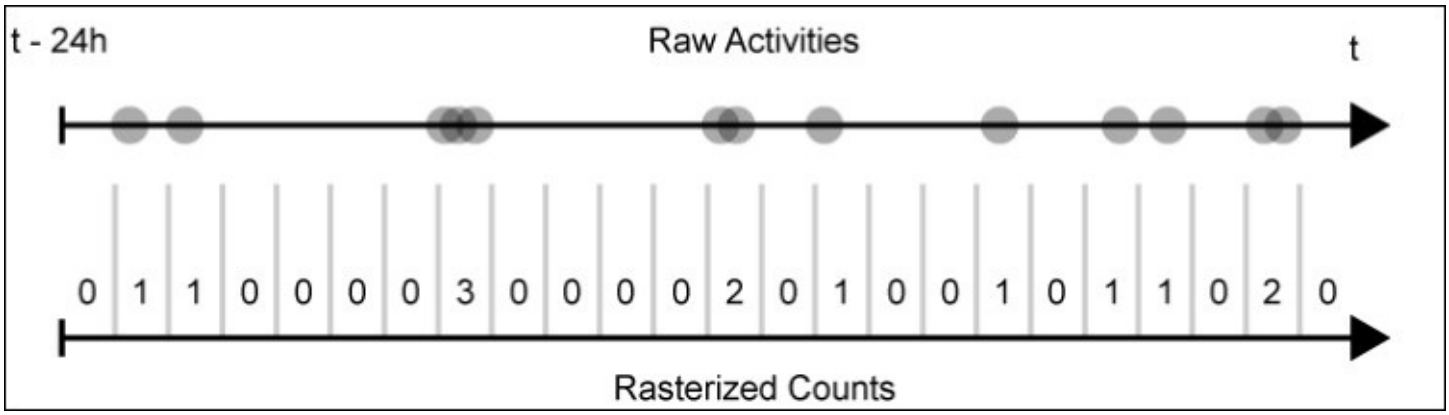
Creating your first chart

In this section, we will create our first chart using Chartist to provide a project activity overview over the past 24 hours. This bar chart will only provide some visual clues about the project activity, and our goal is not to make it provide detailed information. For this reason, we will configure it to hide any labels, scales, and grid lines. The only visible part should be the bars of the bar chart.

Before we start creating the activity chart itself, we need to look at how we need to transform and prepare our data for the charts.

Let's look at what data we already have in our system. As far as the activities go, they all have a timestamp on them stored in the `time` field. However, for our chart, we want something else displayed. What we're looking for is a chart that displays one bar for each hour of the past 24 hours. Each bar should represent the count of activities in that timeframe.

The following illustration shows our source data, which is basically a time stream of activity events. On the lower arrow, we see the data that we need to end up with for our chart:



An illustration displaying activities as a time stream where the dots represent activities. The lower arrow is showing a rasterized count by hour for the last 24 hours.

Let's implement a function that does the transformation outlined in this image. We'll add this function to our `time-utilities` module on the `lib/utilities/time-utilities.js` path:

```
function rasterize(timeData, timeFrame, quantity, now, fill = 0) {
  // Floor to a given time frame
now = Math.floor(now / timeFrame) * timeFrame;
returntimeData.reduce((out, timeData) => {
    // Calculating the designated index in the rasterized output
const index = Math.ceil((now - timeData.time) / timeFrame);
    // If the index is larger or equal to the designed rasterized
    // array length, we can skip the value
if (index < quantity) {
```

```
out[index] = (out[index] || 0) + timeData.weight;
    }
return out;
  }, Array.from({length: quantity}).fill(fill)).reverse();
}
```

Let's look at the input parameters of our newly-created function:

- `timeData`: This parameter is expected to be an array of objects that contains a `time` property that is set to the timestamp of the event that should be counted. The objects should also contain a `weight` property, which is used to count. Using this property, we can count one event as two or even count minus values to decrease the count in a raster.
- `timeFrame`: This parameter specifies the time span of each raster in milliseconds. If we want to have 24 rasterized frames, each consisting of one hour this parameter needs to be set to 3,600,000 (1 h = 60 min = 3,600 s = 3,600,000 ms).
- `quantity`: This parameter sets the amount of rasterized frames that should be present in the output array. In the case of 24 frames of one hour, this parameter should be set to 24.
- `now`: This is when our function is rasterizing time, starting at a given point in time backwards. The `now` parameter sets this point in time.
- `fill`: This is how we can specify how we'd like our rasterized output array to be initialized. In the case of our activity counts, we want this to be set to `0`.

The function that we just created is necessary to create the activity chart. The transformation helps us prepare project activities for the input data of the chart.

It's time to create our first chart component! Let's start with a new template created on the `lib/projects-dashboard/project-summary/activity-chart/activity-chart.html` path:

```
<div #chartContainer></div>
```

As we leave all the rendering up to Chartist, this is actually already all that we need. Chartist needs an element as a container to create the chart in. We set a `chartContainer` local view reference so that we can reference it from our component, and then pass it to Chartist.

Let's move on with the chart creation, and flesh out the activity chart component by creating the `Component` class in `activity-chart.js` in the same directory as the template:

```
...
import Chartist from 'chartist';
import {rasterize, UNITS} from '../../../utilities/time-utilities';

@Component({
selector: 'ngc-activity-chart',
  ...
})
export class ActivityChart {
@Input() activities;
@ViewChild('chartContainer') chartContainer;

ngOnChanges() {
```

```
this.createOrUpdateChart();
  }

ngAfterViewInit() {
this.createOrUpdateChart();
  }
  ...
}
```

Note

Chartist is available for almost all package managers, and it also comes bundled in the **UMD** module format (**Universal Module Format**), which, in fact, is a wrapper to enable **AMD** (**Asynchronous Module Definition**), CommonJS module format, and global export.

Using JSPM, we can simply install Chartist by executing the following command on the command line:

```
jspm install chartist
```

After installing Chartist, we can directly import it using ES6 module syntax.

We also import the rasterize function that we created so that we can use it later to convert our activities into the expected input format for our chart.

As we rely on a view child as a container element to create our chart, we need to wait for the `AfterViewInit` lifecycle hook in order to construct the chart. At the same time, we need to rerender the chart if the input `activities` change. Using the `OnChanges` lifecycle hook, we can react on input changes and update our chart.

Let's now look at the `createOrUpdateChart` function, which does exactly what its name already implies:

```
createOrUpdateChart() {
if (!this.activities || !this.chartContainer) {
return;
  }

consttimeData = this.activities.map((activity) => {
return {
time: activity.time,
weight: 1
    };
  });

const series = [
rasterize(
timeData,
UNITS.find((unit) => unit.short === 'h').milliseconds,
      24,
      +new Date())
  ];
```

```
if (this.chart) {
this.chart.update({ series });
  } else {
this.chart = new Chartist.Bar(this.chartContainer.nativeElement, {
series
    }, {
width: '100%',
height: 60,
axisY: {
onlyInteger: true,
showGrid: false,
showLabel: false,
offset: 0
      },
axisX: {
showGrid: false,
showLabel: false,
offset: 0
      },
chartPadding: 0
    });
  }
}
```

Let's look into the code in more detail and walk through it step by step:

- As we get called both from the `AfterViewInit` and `OnChanges` lifecycle, we need to make sure that both the `chartContainer` and `activities` inputs are ready before we continue.
- Now, it's time to convert the activity data that we receive as input into the rasterized form that is required for the chart that we'd like to create. We use `Array.prototype.map` to transform our activities into the `timeData` objects that are required by the `rasterize` function. We also pass the necessary parameters so that the function will rasterize into 24 frames, each consisting of one hour.
- If the `chart` member is already set to a chart that was previously created, we can use the `update` function on the Chartist chart object to only update with the new data.
- If there's no chart object already, we need to create a new chart. As a first parameter to the `Chartist.Bar` constructor, we'll pass the DOM element reference of our container view child. Chartist will create our chart in this container element. The second argument is our data, which we fill with the series that was just generated. In the chart options, we'll set everything to achieve a very plain-looking chart without any detailed information.

This is great! We created our first chart component using Chartist! Now, we can go back to our `ProjectSummary` component and integrate the activity chart there to provide an activity overview:

```
...
import {ActivityService} from '../../activities/activity-service/activity-
service';
import {ActivityChart} from './activity-chart/activity-chart';

@Component({
```

```
selector: 'ngc-project-summary',
  ...
directives: [EffortsTimeline, ActivityChart],
  ...
})
export class ProjectSummary {
  ...
constructor(@Inject(ActivityService) activityService) {
this.activityService = activityService;
  }

ngOnChanges() {
if (this.project) {
      ...

this.activities = this.activityService.change
        .map((activities) => activities.filter((activity) => activity.subject
=== this.project._id));
    }
  }
}
```

The first change here is to include the `ActivityService` so that we can extract the required project activities to pass them to the `ActivityChart` component. We also need to import the `ActivityChart` component and declare it as a directive on the component.

As our component relies on the project to be provided as input, which is subject to change, we need to implement the logic to extract activities in the `OnChanges` lifecycle hook of the component.

Before we pass on the observable activities stream, we need to filter the activities that come through the stream so that we only get activities that are relevant to the current project. Again, we will use the `async` pipe in order to subscribe to the activities so that there's no need to use a subscribe form within the component. The `activities` property will be directly set to a filtered `Observable`.

Let's look at the changes in the view of the `ProjectSummary` component in order to enable our chart:

```
...
<div class="project-summary__label">Activity last 24 hours</div>
<ngc-activity-chart [activities]="activities | async">
</ngc-activity-chart>
```

We add our `ActivityChart` component at the bottom of the already existing template. We also create the necessary binding to pass our activities into the component. Using the `async` pipe, we can resolve the observable stream and pass the filtered activities list into the `chart` component.

Finally, our `ProjectSummary` component looks great and immediately provides a project insight by displaying the aggregated efforts timeline and a nice activity chart. In the next

section, we'll dive a bit deeper into the charting capabilities of Chartist, and we will also provide some interactivity using Angular.

Visualizing open tasks

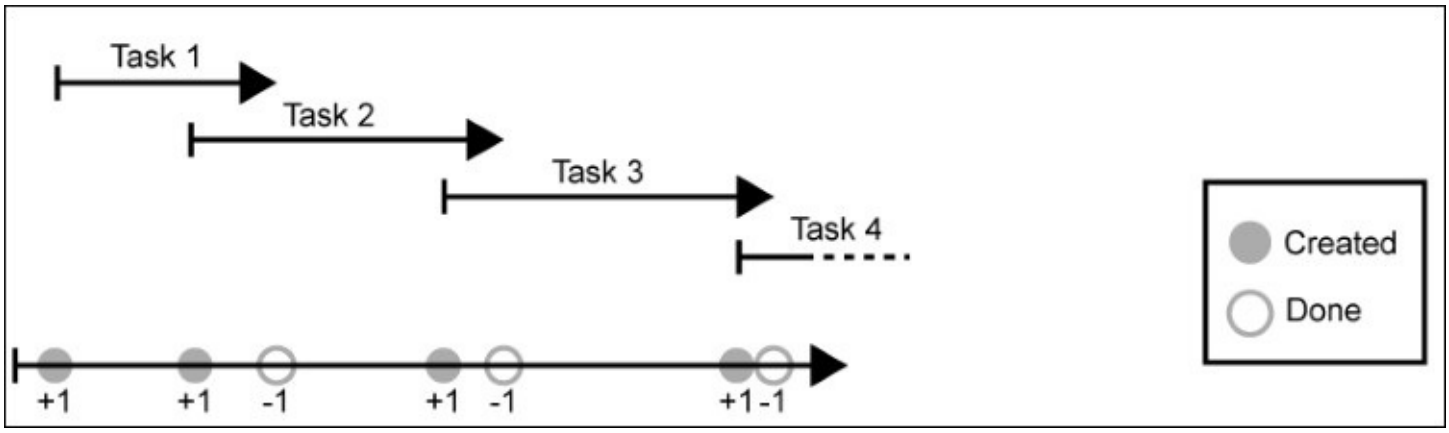
In this section, we will create a chart component using Chartist, which will display the open task progress of projects over time. To do this, we'll use a line chart with a specific interpolation that provides quantized steps rather than lines with directly connected points.

We are also providing some interactivity in that the user will be able to switch the displayed time frame using a toggle button. This allows us to reuse the `Toggle` component that we created in Chapter 2, *Ready, Set, Go!*, of this book.

Let's first look at the data that we have in our system and how we can transform it into the data needed by Chartist.

We can rely on two data attributes of our tasks in order to draw them onto a timeline. The `created` attribute is set to the timestamp at the moment when the task was created. If a task is marked as done, the `done` attribute is set to the timestamp at that time.

As we're only interested in the amount of open tasks at any given time, we can safely presume a model where we put all tasks onto a single timeline and where we are only concerned about the `created` and `done` timestamps as events. Let's look at the following illustration to get a better understanding of the problem:

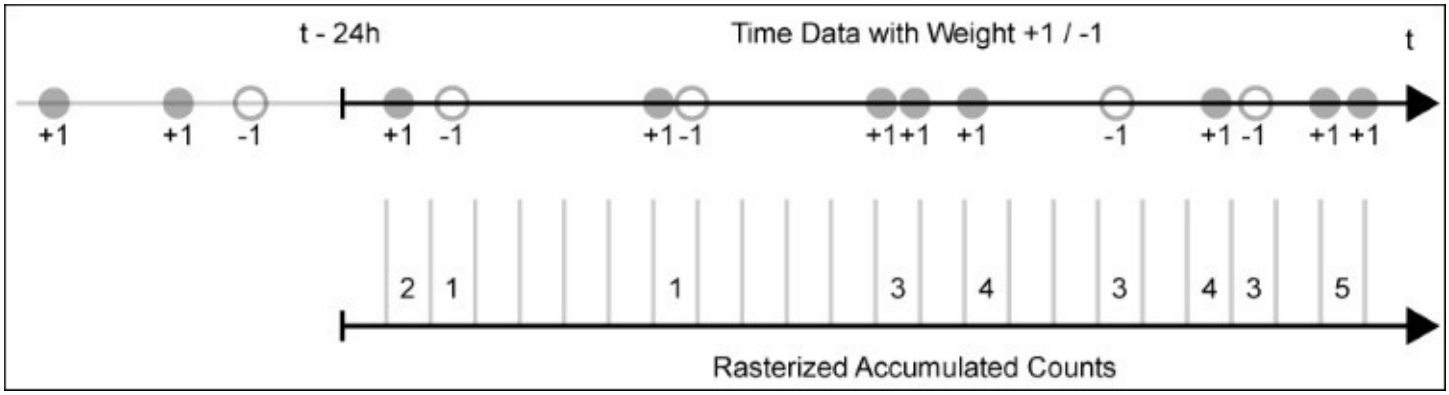


An illustration that shows how we can represent all task timelines on a single timeline using the created and done events. The created events count as a +1, while the done events count as -1.

The lower arrow is a representation of all tasks of the `created` and `done` events on a timeline. We can now use this information as input to our `rasterize` function in order to get the data that we need for our chart. As the `timeData` objects that are used as input for the rasterization also support a `weight` property, we can use this to represent the `created (+1)` or `done (-1)` events.

We need to make a slight modification to our rasterize function. So far, the rasterize function only counts events together in frames. However, for the open task counts, we look into an accumulation over time. If the task count changes, we need to keep the value until it changes again. In transformation of activities in the previous topic, we didn't use this same logic. There, we only counted events inside frames, but there was no accumulation.

Let's look at the following illustration to see the difference as compared to the rasterization that we applied while processing activities:



An illustration that shows how we can accumulate the open tasks count over time

We can count each `weight` property of the `timeData` objects (events) together over time. Only if there's a change of the accumulated value, we will write the current accumulated value into the rasterized output array.

Let's open our `time-utilities` module and apply the changes to the `rasterize` function:

```
export function rasterize(timeData, timeFrame, quantity,
now = +new Date(), fill = 0,
accumulate = false) {
  // Floor to a given time frame
now = Math.floor(now / timeFrame) * timeFrame;
  // Accumulation value used for accumulation mode to keep track
  // of current value
let accumulatedValue = 0;

  // In accumulation mode we need to be sure that the time data
  // is ordered
if (accumulate) {
timeData = timeData.slice().sort(
      (a, b) => a.time < b.time ? -1 : a.time > b.time ? 1 : 0);
  }

return timeData.reduce((rasterized, timeData) => {
    // Increase the accumulated value, in case we need it
accumulatedValue += timeData.weight;
    // Calculating the designated index in the rasterized output
    // array
```

```
const index = Math.ceil((now - timeData.time) / timeFrame);
    // If the index is larger or equal to the designed rasterized
    // array length, we can skip the value
if (index < quantity) {
rasterized[index] = accumulate ?
accumulatedValue :
        (rasterized[index] || 0) + timeData.weight;
    }
return rasterized;
  }, Array.from({length: quantity}).fill(fill)).reverse();
}
```

Let's walk through the changes that we applied to the `rasterize` function to allow accumulation of frames:

- First of all, we added a new parameter to our function with the name `accumulate`. We used the ES6 default parameters to set the parameter to `false` if no value was passed into the function when called.
- We now define a new `accumulatedValue` variable, which we initialize with `0`. This variable will be used to keep track of the sum of all `weight` values over time.
- The next bit of code is very important. If we want to accumulate the sum of all `weight` values over time, we need to make sure that these values come in sequence. In order to ensure this, we sort the `timeData` list by its items `time` attribute.
- In the `reduce` callback, we increase the `accumulatedValue` variable by the `weight` value of the current `timeData` object.
- If the `timeData` object falls into a rasterized frame, we do not increase this frame's count like we did before. In accumulation mode, we set the frames count to the current value in `accumulatedValue`. This will result in all changed accumulated values being reflected in the rasterized output array.

This is all the preparation that we need to process the date in order to render our open tasks chart. Let's move on and create the `Component` class of our new `chart` component.

Creating an open tasks chart

In the following component, we will utilize the refactored `rasterize` function of the previous topic. Using the new accumulate function, we can now track open task counts over time.

Let's start with the `Component` class in a new `lib/projects-dashboard/tasks-chart/tasks-chart.js` file to implement our `Component` class:

```
...
import Chartist from 'chartist';
import Moment from 'moment';

import {rasterize} from '../../utilities/time-utilities';

@Component({
selector: 'ngc-tasks-chart',
  ...
})
export class TasksChart {
@Input() projects;
@ViewChild('chartContainer') chartContainer;

ngOnChanges() {
this.createOrUpdateChart();
  }

ngAfterViewInit() {
this.createOrUpdateChart();
  }
    ...
}
```

So far, this looks exactly like our first `chart` component where we visualized project activities. We also imported Chartist as we will use it to render our chart in the `createOrUpdateChart` function that we'll create shortly. The chart that we will create will contain much more detailed information. We will render both axis labels and some scales. In order to format our labels that basically contain timestamps, we use the Moment.js library once again.

We also use the `projects` input data and transform it with the amended `rasterize` utility function in order to prepare all the data for our line chart.

Let's move on and flesh out the `createOrUpdateChart` method of our component:

```
createOrUpdateChart() {
if (!this.projects || !this.chartContainer) {
return;
  }

  // Create a series array that contains one data series for each
  // project
const series = this.projects.map((project) => {
```

```
    // First we need to reduces all tasks into one timeData list
const timeData = project.tasks.reduce((timeData, task) => {
      // The created time of the task generates a timeData with
      // weight 1
timeData.push({
time: task.created,
weight: 1
      });
      // If this task is done, we're also generating a timeData
      // object with weight -1
if (task.done) {
timeData.push({
time: task.done,
weight: -1
        });
      }
return timeData;
    }, []);

    // Using the rasterize function in accumulation mode, we can
    // create the required data array that represents our series
    // data
return rasterize(timeData, 600000, 144, +new Date(),
null, true);
  });

const now = +new Date();
  // Creating labels for all the timeframes we're displaying
const labels = Array.from({
length: 144
  }).map((e, index) => now - index * 600000).reverse();

if (this.chart) {
    // If we already have a valid chart object, we can simply
    // update the series data and labels
this.chart.update({
series,
labels
    });
  } else {
    // Creating a new line chart using the chartContainer element
    // as container
this.chart = new Chartist.Line(this.chartContainer.nativeElement, {
series,
      labels
    }, {
      width: '100%',
height: 300,
      // Using step interpolation, we can cause the chart to
      // render in steps instead of directly connected points
lineSmooth: Chartist.Interpolation.step({
        // The fill holes setting on the interpolation will cause
        // null values to be skipped and makes our line to
        // connect to the next valid value
fillHoles: true
      }),
```

```
axisY: {
onlyInteger: true,
low: 0,
offset: 70,
        // We're using the label interpolation function for
        // formatting our open tasks count
labelInterpolationFnc: (value) => `${value} tasks`
      },
axisX: {
        // We're only displaying two x-axis labels and grid lines
labelInterpolationFnc: (value, index, array) => index % (144 / 4) === 0 ?
Moment(value).calendar() : null
      }
    });
  }
}
```

Okay, that's quite a bit of code here. Let's walk through it step by step to gain a better understanding of what's going on:

1. First, we need to create our transformed series data by mapping the projects list. The series array should include one data array for each project. Each data array will contain the open project tasks over time.
2. As the `rasterize` function expects a list of `timeData` objects, we first need to transform the projects task list into this format. By reducing the task list, we create a single list of the `timeData` objects. The `reduce` function callback will generate one `timeData` object with a `weight` of 1 for each task. Additionally, it will generate a `timeData` object for each task marked as one with the `weight` value -1. This will result in the desired `rimeData` array, which we can use to accumulate and rasterize.
3. After preparing the `timeData` list, we can call the `rasterize` function in order to create a list of open tasks over a certain amount of timeframes. We use a 10 minute timeframe (600000ms) and rasterize this with 144 frames. This makes a total of 24 hours.
4. Besides the series data, we will also need labels for our chart. We create a new array and initialize this with 144 timestamps, all of which are set to the start of the 144 rasterized frames that we display on the chart.
5. Now, we have the series data and the labels ready, and all that's left to do is to render our chart.
6. Using the `lineSmooth` configuration, we can specify a special kind of interpolation for our line chart. The step interpolation will not connect each point in our line chart directly, but rather it will move in discrete steps to move from point to point. This is exactly what we're looking for to render the open task counts over time.
7. Setting the `fillHoles` option to `true` in the step interpolation is very important. Using this setting, we can actually tell Chartist that it should close any gaps within the data (actually null values) and connect the line to the next valid value. Without this setting, we'd see gaps on the chart between the task count changes in our data arrays.
8. One last important thing in our code is the `labelInterpolationFnc` option that we set on the *x* axis configuration. This function can not only be used to format a label or interpolate any expression that may come along with the label, but it also allows us to

return null instead. Returning null from this function will cause Chartist to skip the given label and the corresponding grid line. This is very useful if we'd like to skip certain labels by their value or by the index of the label. In our code, we ensure that we only render four labels of all 144 generated labels.

Let's take a look at the rather simple template of our component in the `tasks-chart.html` file in the same folder as our `Component` class file:

```
<div #chartContainer class="tasks-chart__container"></div>
```

The same as with the `ActivityChart` component, we only create a simple chart container element, which we already reference in our `Component` class.

This is basically all that we needed to do in order to create an open tasks chart using Chartist. However, there's still some room for improvement here:

Open tasks visualized with our tasks chart component using Chartist's step interpolation

Creating a chart legend

Currently, there's no way to tell exactly which of the lines represents what project. We see one colored line for each project, but we can't associate these colors. What we need is a simple legend that helps our users to associate line chart colors to projects.

Let's look at the required code changes to implement legends on our chart. In the `Component` class of our `TasksChart` component, we need to perform the following modifications:

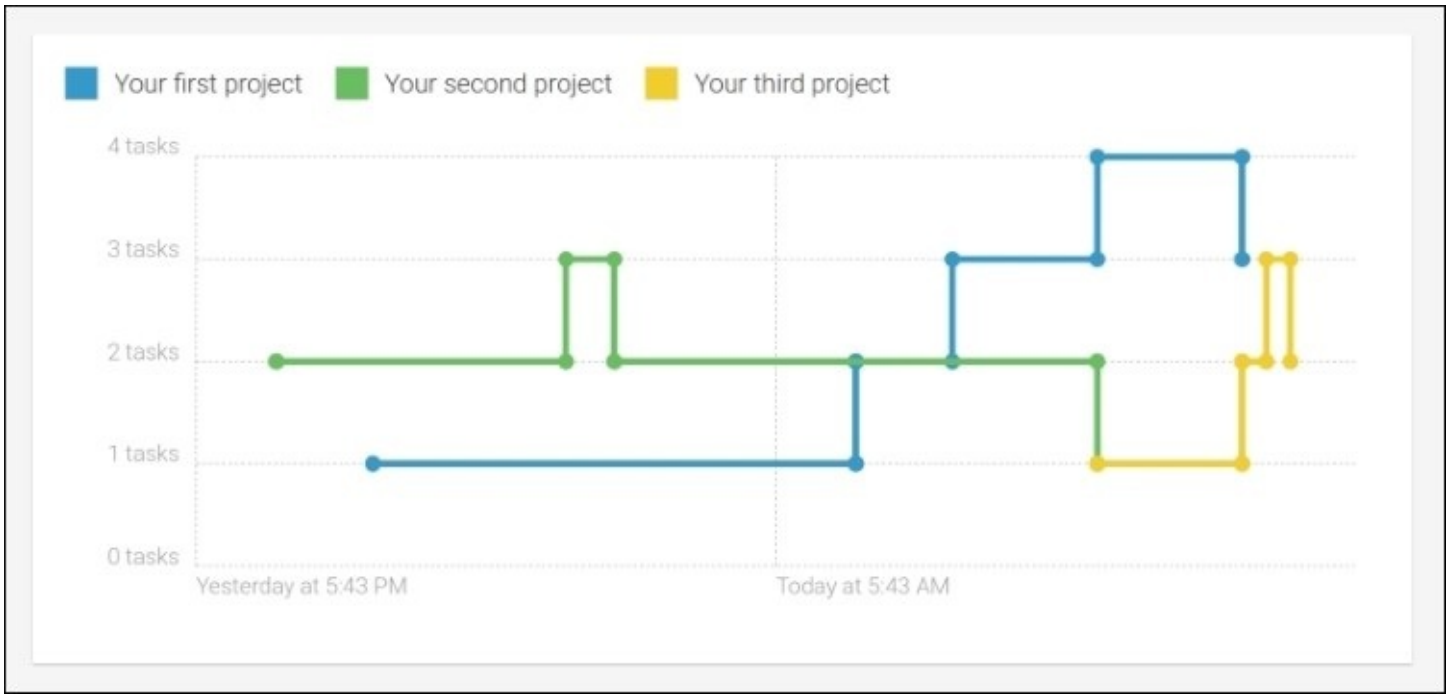
```
...
export class TasksChart {
  ...
ngOnChanges() {
if (this.projects) {
      // On changes of the projects input, we need to update the
      // legend
this.legend = this.projects.map((project, index) => {
return {
name: project.title,
class: `tasks-chart__series--series-${index + 1}`
        };
      });
    }

this.createOrUpdateChart();
  }
  ...
}
```

In the `OnChanges` lifecycle hook, we map the projects input to a list of objects that contain a `name` and `class` property, which will support us in rendering a simple legend. The template `` `tasks-chart__series--series-${index + 1}` `` string will generate the necessary class name to render the right color into our legend.

Using this legend information, we can now go ahead and implement the necessary template changes to render the legend in our `chart` component:

```
<ul class="tasks-chart__series-list">
<li *ngFor="let series of legend"
class="tasks-chart__series {{series.class}}">
    {{series.name}}
</li>
</ul>
<div #chartContainer class="tasks-chart__container"></div>
```

Well, that was a piece of cake, right? However, the result speaks for itself. We created a nice legend for the chart in just a couple of minutes:



Open tasks chart with our added legend

Making tasks chart interactive

Currently, we hardcoded the timeframe of our open task chart to be 144 frames, each of 10 minutes, making a total of 24 hours displayed to the user. However, maybe our users would want to change this view.

In this topic, we will create a simple input control using our `Toggle` component, which allows our users to change the timeframe settings of the chart.

We will provide the following views as options:

- **Day:** This view will rasterize 144 frames, each consisting of 10 minutes, which makes a total of 24 hours
- **Week:** This view will rasterize 168 frames, each consisting of one hour, which makes a total of seven days
- **Year:** This view will rasterize 360 frames, each representing a full day

Let's start with the implementation of our timeframe switch by modifying the `Component` class code of the `TasksChart` component:

```
...
import {Toggle} from '../../ui/toggle/toggle';

@Component({
  ...
directives: [Toggle]
})
export class TasksChart {
  ...
constructor() {
    // Define the available time frames within the chart provided
    // to the user for selection
this.timeFrames = [{
name: 'day',
timeFrame: 600000,
amount: 144
    }, {
name: 'week',
timeFrame: 3600000,
amount: 168
    }, {
name: 'year',
timeFrame: 86400000,
amount: 360
    }];
    // From the available time frames, we're generating a list of
    // names for later use within the Toggle component
this.timeFrameNames
      = this.timeFrames.map((timeFrame) => timeFrame.name);
    // The currently selected timeframe is set to the first
    // available one
this.selectedTimeFrame = this.timeFrames[0];
```

```
  }
  ...
createOrUpdateChart() {
    ...
const series = this.projects.map((project) => {
      ...
return rasterize(timeData,
this.selectedTimeFrame.timeFrame,
this.selectedTimeFrame.amount,
                        +new Date(), null, true);
    });

const now = +new Date();
const labels = Array.from({
length: this.selectedTimeFrame.amount
    }).map((e, index) => now - index *
this.selectedTimeFrame.timeFrame).reverse();
  ...
  }
  ...
  // Called from the Toggle component if a new timeframe was
  // selected
onSelectedTimeFrameChange(timeFrameName) {
    // Set the selected time frame to the available timeframe with
    // the name selected in the Toggle component
this.selectedTimeFrame =
this.timeFrames.find((timeFrame) =>
timeFrame.name === timeFrameName);
this.createOrUpdateChart();
  }
}
```

Let's go through these changes briefly:

1. First of all, we added a constructor to our `Component` class in which we initialized three new members. The `timeFrames` member is set to an array of timeframe description objects. They contain the `name`, `timeFrame`, and `amount` properties, which are later used for the calculations. The `timeFrameNames` member contains a list of timeframe names, which is directly derived from the `timeFrames` list. Finally, we have a `selectedTimeFrame` member, which simply points to the first available timeframe object.
2. In the `createOrUpdateChart` function, we no longer rely on hardcoded values for the task count rasterization, but we refer to the data in the `selectedTimeFrame` object. By changing this object reference and calling the `createOrUpdateChart` function again, we can now switch the view on the underlying data dynamically.
3. Finally, we added a new `onSelectedTimeFrameChange` method, which acts as a binding to the `Toggle` component, and this will be called whenever the user selects a different timeframe.

Let's look at the necessary template changes to enable switching of timeframes:

```
<ngc-toggle
      [buttonList]="timeFrameNames"
```

```
      [selectedButton]="selectedTimeFrame.name"
      (selectedButtonChange)="onSelectedTimeFrameChange($event)">
</ngc-toggle>
...
<div #chartContainer class="tasks-chart__container"></div>
```

From the bindings to the `Toggle` component, you can already tell that we rely on the `timeFrameNames` member on our component to represent all selectable timeframes. We also bind to the `selectedButton` input property of the `Toggle` component using the `selectedTimeFrame.name` property. On changes of the selected button in the `Toggle` component, we call the `onSelectedTimeFrameChange` function, where the timeframe is switched and the chart is updated.

This is all that we need to enable switching the timeframe on our chart. The user can now choose between the year, week, and day views.

Our `TasksChart` component is now ready to be integrated into our dashboard. We can achieve this with some small changes to the template of our `ProjectsDashboard` component:

```
...
<div class="projects-dashboard__l-main">
<h3 class="projects-dashboard__sub-title">Tasks Overview</h3>
<div class="projects-dashboard__tasks">
<ngc-tasks-chart [projects]="projects | async">
</ngc-tasks-chart>
</div>
  ...
</div>
```

This is basically all that we need to make, and after this change, our dashboard contains our nice chart displaying open task counts over time.

In the binding of the `TasksChart` projects input property, we use the `async` pipe once again to resolve the observable stream of projects directly in the view.

Summary

In this chapter, we learned about Chartist and how to use it in conjunction with Angular to create good looking and functional charts. We can leverage the power of both worlds, and create reusable chart components that are nicely encapsulated.

Just like in most real cases, we always have a lot of data, but the one that we need in a particular case. We learned how we can transform existing data into a form that is optimized for visual representation.

In the next chapter, we will look at building a plugin system in our application. This will allow us to develop portable functionality that is packaged into plugins. Our plugin system will load new plugins dynamically, and we will use it to develop a simple agile estimation plugin.

Chapter 10. Making Things Pluggable

I'm a huge fan of plugin architectures. Besides their tremendously positive effect on your application and scope management, they are also a lot of fun to develop. I'd recommend integrating a plugin architecture in their library or application to anyone who asks me. A good plugin architecture allows you to write a concise application core and provide additional functionality via plugins.

Designing your whole application in a way that it allows you to build a plugin architecture has a great effect on the extensibility of your system. This is because you're making your application open for extensibility but closing it for modification.

While authoring my open source projects, I also experienced that a plugin architecture helps you manage the scope of your project. Sometimes, a requested feature is really nice and helpful, but it will still bloat the library core. Instead of bloating your whole application or library with such features, you can simply write a plugin to get the job done.

In this chapter, we will create our own plugin architecture that will help us extend the features of our application without bloating its core. We'll first build the plugin API in the core of our application and then use the API to implement a nice little agile plugin, which helps us to estimate tasks using story points.

We'll cover the following topics in this chapter:

- Designing a plugin architecture, based on the Angular ecosystem
- Implementing a decorator-based plugin API
- Using `ComponentResolver` and `ViewContainerRef` to instantiate plugin components into predefined slots in our application
- Implementing a plugin-loading mechanism using SystemJS
- Using a reactive approach in our plugin architecture to enable plug and play style plugins
- Implementing an agile plugin to record story points using the new plugin API

Plugin architecture

At a higher level, a plugin architecture should fulfil at least the following requirements:

- **Extensibility**: The main idea behind plugins is to allow the extension of the core functionality using isolated bundles of code. A great plugin architecture allows you to extend the core seamlessly and without noticeable performance losses.
- **Portability**: Plugins should be isolated enough so that they can be plugged into the system during runtime. There shouldn't be a necessity to rebuild a system to enable plugins. Ideally, plugins can even be loaded at any time during runtime. They can be deactivated and activated and should not cause the system to run into an inconsistent state.
- **Composability**: A plugin system should allow the use of many plugins in parallel and allow an extension of the system by compositing multiple plugins together. Ideally, the system also includes dependency management, plugin version management, and plugin intercommunication.

There are a lot of different approaches on how to implement a plugin architecture. Although these approaches can vary a lot, there's almost always a mechanism in place that provides unified extension points. Without this, it will be hard to extend a system uniformly.

I've worked with some plugin architectures in the past, and besides using existing plugin mechanisms, I've also enjoyed designing some of them myself. The following list should provide an idea about some of the approaches that you can use when designing a plugin system:

- **DSL**: Using domain-specific languages is one way to implement a pluggable architecture. After you've implemented the core of your application, you can develop an API or even a scripting language that allows you to develop further features using this DSL. A lot of video game engines and CG applications rely on this approach. Although this approach is very flexible, it can also lead to performance issues quickly, and it's prone to introducing complexity. Mostly, the prerequisites to implement such an architecture are to expose very low-level core operations (such as adding UI elements, configuring process flows, and so on) into the DSL, which does not provide clear boundaries and extension points but is extremely flexible. Some examples of DSL-based plugin systems are most of Adobe's CG applications, 3D Studio Max, and Maya, but also game engines, such as Unreal Engine or the Real Virtuality Engine from Bohemia Interactive Studio.
- **The core is the plugin system**: Another approach is to build such a sophisticated plugin architecture that it fulfils all the outlined requirements in the previous listing (extensibility, portability, and composability) and even some more sophisticated requirements on top. The core of your application is one large plugin system. Then, you start to implement everything as a plugin. Even the core concerns of your application will be implemented as plugins. A perfect example of this approach is the Eclipse IDE with its Equinox core. The problem with this approach is that you're likely to run into performance problems as your application grows. As everything is a plugin,

optimization is quite tricky, and plugin compatibility can make the application unstable.

- **Event-based extension points**: Also, a great way to provide extensibility of a system is by opening up the pipeline of your system to input from outside. Imagine that for every important step in your application, you notify the outside world about the step and allow interception before the application continues with processing. In this manner, a plugin will just be an adapter that listens for these pipeline events of your application and then modifies the behavior as required. A plugin itself can also emit events, which then can be processed by other plugins again. This architecture is really flexible, as it allows you to change the behavior of your core functionality without introducing too much complexity. It's also fairly easy to implement this approach even after you've finished your core without any thoughts about a plugin system. I've been following this approach in my open source project Chartist and, so far, I've had very good results with it.
- **Plugin interfaces**: An application can expose a set of interfaces that define certain extension points. This approach is heavily used in the Java framework where it's known as **Service Provider Interface** (**SPI**). Providers implement a certain contract, which allows the core system to rely on an interface rather than an implementation. These providers can then be cycled back into the system where they are made available to the framework and other providers. Although this is probably the safest way to provide extensibility in terms of uniformness, it's also the most rigid one. A plugin will never be allowed to do anything else that was specified in the contract of the interfaces.

You can see that all four approaches vary a lot. From the top-most, which provides extreme flexibility at the cost of complexity and stability, to the bottom-most, which is very robust but also rigid.

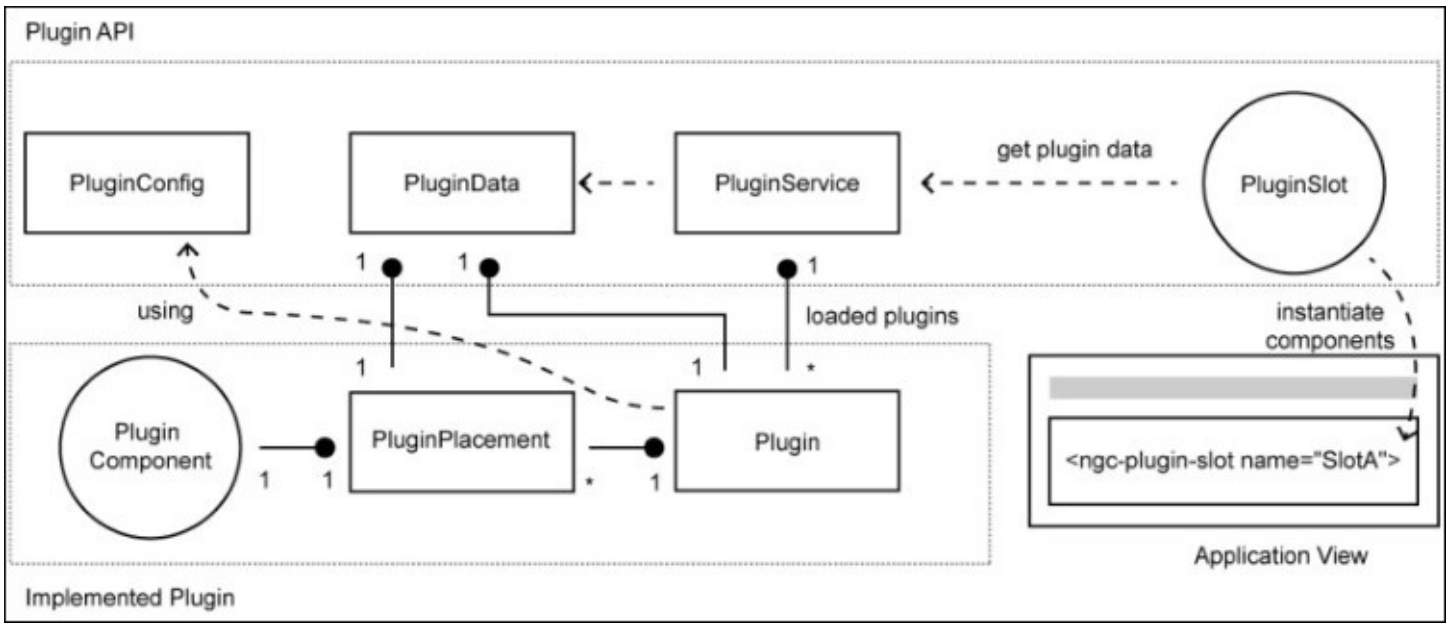
The approach that you choose when implementing a plugin system heavily depends on the requirements for your application. If you do not plan on building an application that comes bundled in various flavors and where multiple versions for completely different concerns should exist, the approaches to the bottom of the preceding listing are probably more likely the ones that you should follow.

Pluggable UI components

The system that we're going to build in this chapter borrows a lot of mechanisms that are already present in the Angular framework. In order to implement extensibility using plugins, we rely on the following core concepts:

- We use directives to indicate extension points in the UI, which we call plugin slots. These plugin slot directives will be responsible for the dynamic instantiation of plugin components and will insert them into the application UI at the given position.
- Plugins expose components using a concept that we call plugin placements. Plugin placements declare what components of a plugin should be placed into which plugin slots in the application. We also use plugin placements to decide the order in which components from different plugins should be inserted into the plugin slots. For this, we'll use a property called **priority**.
- We use the dependency injection of Angular to provide the instantiated plugin information into the plugin components. As the plugin components will be placed in a spot where there's already an injector present, they will be able to inject surrounding components and dependencies in order to connect to the application.

Let's look at the following illustration to picture the architecture of our plugin system before we start implementing it:



The plugin architecture that we'll implement in this chapter using some basic UML and cardinality annotations

Let's look at the different entities in this diagram and quickly explain what they do:

- `PluginConfig`: This ES7 decorator is the key element when implementing a plugin. By

annotating a plugin class using this decorator, we can store meta-information about the plugin, which will be used later by our plugin system. The metadata includes the plugin name, a description, and the placement information.

- `PluginData`: This is an aggregation class that is used by the plugin system to couple the information about an instantiated plugin with the placement information (where plugin components should be instantiated). This entity is exposed in dependency injection once a plugin component is created. Any plugin component can make use of this entity to gather information about the instantiation or to gain access to the plugin instance.
- `PluginService`: This is the service used to glue our plugin system together. It's mainly used to load plugins, remove plugins, or used by the `PluginSlot` directive to gather plugin components together that are relevant for creation in the plugin slot.
- `PluginSlot`: This directive is used to mark UI extension points in our application. Wherever we'd like to make it possible for plugins to hook into our application user interface, we'll place this directive. Plugin slots need to be named, and plugins use placement information to reference slots by their name. This way a plugin can provide different components for different slots in our application.
- `PluginComponent`: These are regular Angular components that come bundled with a plugin implementation. A plugin can provide multiple components configured on the plugin using a `PluginPlacement` object.
- `PluginPlacement`: This is used in the plugin configuration where a plugin can have multiple placement configurations. Each placement entity consist of a reference to a component, the name of the slot where the component should be instantiated, and a priority number that helps the plugin system to order plugin components correctly when multiple components get instantiated in the same slot.
- `Plugin`: This is the actual plugin class when implementing a plugin. The class contains the plugin configuration annotated using the `PluginConfig` decorator. The plugin class is instantiated once in the application and is also shared across the plugin components using the dependency injection of Angular. Therefore, this class is also a good place to share data between plugin components.

Now, we have an overview of what we're going to build on a higher level. Our plugin system is very rudimentary, but it will support things such as hot loading plugins (plug and play style) and other nice features. In the next topic, we'll start by implementing the plugin API core components.

Implementing the plugin API

Let's start with the less complex entities of our plugin API. We create a new `lib/plugin/plugin.js` file to create the `PluginConfig` decorator and the `PluginPlacement` class, which stores the information where plugin components should be placed. We also create the `PluginData` class in this file, which is used to inject plugin runtime information into plugin components:

```
export function PluginConfig(config) {
  return (type) => {
    type._pluginConfig = config;
  };
}
```

The `PluginConfig` decorator contains the very simple logic of accepting a configuration parameter, which will then be stored on the annotated class (the constructor function) on the `_pluginConfig` property. If you need a refresher on how decorators work, it's maybe a good time to read the decorator topic in Chapter 1, *Component-Based User Interfaces*, again:

```
export class PluginPlacement {
  constructor(options) {
    this.slot = options.slot;
    this.priority = options.priority;
    this.component = options.component;
  }
}
```

The `PluginPlacement` class represents the configuration object to expose plugin components into different plugin slots in the application UI:

```
export class PluginData {
  constructor(plugin, placement) {
    this.plugin = plugin;
    this.placement = placement;
  }
}
```

The `PluginData` class represents the plugin runtime information that was created during instantiation of the plugin as well as one `PluginPlacement` object. This class will be used by the `PluginService` to convey information about plugin components to the plugin slots in the application.

These three classes are the main interaction points when implementing a plugin.

Let's look at a simple example plugin, to get a picture of how we can use the `PluginConfig` decorator and the `PluginPlacement` class to configure a plugin:

```
@PluginConfig({
  name: 'my-example-plugin',
  description: 'A simple example plugin',
```

```
  placements: [
    new PluginPlacement({
      slot: 'plugin-slot-1',
      priority: 1,
      component: PluginComponent1
    }),
    new PluginPlacement({
      slot: 'plugin-slot-2',
      priority: 1,
      component: PluginComponent2
    })
  ]
})
export default class ExamplePlugin {}
```

Using the `PluginConfig` decorator, implementing a new plugin is a breeze. We decide the name, description, and where we'd like to place plugin components in the application at design time.

Our plugin system uses named `PluginSlot` directives to indicate extension points in our application component tree. In the `PluginPlacement` objects, we reference the Angular components built into the plugin and indicate in which slot they should be placed by referencing the plugin slot name. The priority of the placement will tell the plugin slot how to order the plugin component when created. This gets important when components of different plugins get created in the same plugin slot.

Okay, let's dive right into the core of our plugin architecture by implementing the plugin service. We'll create a new `lib/plugin/plugin-service.js` file and create a new `PluginService` class:

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

@Injectable()
export class PluginService {
  ...
}
```

As we will create an injectable service, we'll annotate our `PluginService` class using the `@Injectable` annotation. We use the RxJS `ReplaySubject` type in order to emit events on any changes of the activated plugins.

Let's look at the constructor of our service:

```
constructor() {
  this.plugins = [];
  // Change observable if the list of active plugin changes
  this.change = new ReplaySubject(1);
  this.loadPlugins();
}
```

First, we initialize a new empty `plugins` array. This will be the list of active plugins, which contains runtime plugin data such as the URL where the plugin was loaded from, the plugin type (constructor of the class), a shortcut to the configuration stored on the plugin (created by the `PluginConfig` decorator) and finally, the instance of the plugin class itself.

We also add a `change` member that we initialize with a new RxJS `ReplaySubject`. We'll use this subject in order to emit the list of active plugins once it changes. This allows us to build our plugin system in a reactive way and enable plug and play style plugins.

As a last action in the constructor, we call the `loadPlugins` method of the service. This will perform the initial loading with the registered plugins:

```
loadPlugins() {
  System.import('/plugins.js').then((pluginsModule) => {
    pluginsModule.default.forEach(      (pluginUrl) =>
this.loadPlugin(pluginUrl)
    );
  });
}
```

The `loadPlugins` method asynchronously loads a file with the name `plugins.js` from the root path of our application using SystemJS. The `plugins.js` file is expected to default export an array, which contains preconfigured paths to plugins that should be loaded with the application startup. This allows us to configure the plugins that we're already aware of and which should be present by default. Using a separate and asynchronously loaded file for this configuration gives us a better separation from the main application. We can run the same application code but using a different `plugins.js` file and control what plugins should be present by default.

The `loadPlugins` method then loads each plugin using the URL present in the `plugins.js` file by calling the `loadPlugin` method:

```
loadPlugin(url) {
  return System.import(url).then((pluginModule) => {
    const Plugin = pluginModule.default;
    const pluginData = {
      url,
      type: Plugin,
      // Reading the meta data previously stored by the @Plugin
      // decorator
      config: Plugin._pluginConfig,
      // Creates the plugin instance
      instance: new Plugin()
    };

    this.plugins = this.plugins.concat([pluginData]);
    this.change.next(this.plugins);
  });
}
```

The `loadPlugin` method is responsible for the loading and instantiation of individual plugin

modules. It will take the URL of a plugin module as parameter and uses `System.import` to dynamically load the plugin module. The benefits we get from using `System.import` for this job is that we can load both, already existing modules in the bundled application as well as remote URL's using HTTP requests. This makes our plugin system very portable, and we can even load modules during runtime from a different server, from NPM or even GitHub. Of course, SystemJS also supports different module formats, such as ES6 modules or CommonJS modules, as well as different transpilers if the modules are not already transpiled.

After the plugin module is successfully loaded, we bundle all information about the loaded plugin together into a `pluginData` object. We can then add this information to our `plugins` array and emit a new event on our `ReplaySubject` to notify interested parties about the change.

Finally, we'll need a method to gather the `PluginPlacement` data from all our plugins and filter them by a slot name. This gets important when our plugin slots need to know which components they should instantiate. Plugins can expose any number of components into any number of application plugin slots. This function will be used by our plugin slots when they need to know which of the exposed Angular components are relevant to them:

```
getPluginData(slot) {
  return this.plugins.reduce((components, pluginData) => {
    return components.concat(
     pluginData.config.placements
      .filter((placement) => placement.slot === slot)
      .map((placement) => new PluginData(pluginData, placement))
    );
  }, []);
```

This is already it for the `PluginService` class so far, and we created the core of our plugin system. In the next chapter, we will deal with the plugin slots and look at how we can instantiate plugin components dynamically.

Instantiating plugin components

Now, it's time to look at the second major piece of our plugin architecture, which is the `PluginSlot` directive that is responsible for the instantiation of plugin components in the right spots.

Before we get to implement the directive though, let's look at how we can instantiate a component dynamically in Angular. We already covered instantiating views that can contain components in Chapter 7, *Components for User Experience*. In the infinite scroll directive, we used the `ViewContainerRef` to instantiate template elements. However, we have a different use case here. We'd like to instantiate a single component into an existing view.

The `ViewContainerRef` object also provides us with a solution to this problem. Let's look at a very basic example on how to use the `ViewContainerRef` object to instantiate a component. In the following example, we make use of four new concepts:

- Using `@ViewChild` with `read` options set to `{read: ViewContainerRef}` to query for view container instead of the element
- Using the `ComponentResolver` instance to obtain the factory of the component, which we want to instantiate dynamically
- Using `ReflectiveInjector` to create a new child injector that is used for our instantiated component
- Using `ViewContainerRef.createComponent` to instantiate a component and attach it to the underlying view of the view container.

The following code example shows how we can dynamically create a component using the `ViewContainerRef` instance.

```
import {Component, Inject, ViewChild, ViewContainerRef, ComponentResolver} from
'@angular/core';

@Component({
  selector: 'hello-world',
  template: 'Hello World'
})
export class HelloWorld {}

@Component({
  selector: 'app'
  template: '<h1 #headingRef>App</h1>'
})
export class App {
  @ViewChild('headingRef', {read: ViewContainerRef}) viewContainer;

  constructor(@Inject(ComponentResolver) resolver) {
    this.resolver = resolver;
  }

  ngAfterViewInit() {
    this.resolver
```

```
      .resolveComponent(HelloWorld)
      .then((componentFactory) => {
        this.viewContainer.createComponent(componentFactory);
      });
  }
}
```

Injected into the constructor of the `App` component, we can later use `ComponentResolver` to resolve the `HelloWorld` component. We use the `@ViewChild` decorator to query for the heading element in the `App` component. Usually, this would give us the `ElementRef` object that is associated with the view element. However, as we need the view container associated with the element, we can use the `{read: ViewContainerRef}` options to obtain the `ViewContainerRef` object instead.

In the `AfterViewInit` lifecycle hook, we first call the `resolveComponent` method on the `ComponentResolver` instance. This call returns a promise, which resolves to an object of the `ComponentFactory` type. Angular uses component factories internally in order to create components.

After the promise has been resolved, we can now use the `createComponent` method on the view container of our heading element to create our `HelloWorld` component.

Let's look at the `createComponent` method of the `ViewContainerRef` object in more detail:

Method	**Description**
`ViewContainerRef.createComponent`	This method will create a component that is based on the component factory provided in the `componentFactory` parameter. The compiled component will then be attached to the view container at a specific position provided by the `index` parameter. The following are the parameters: • `componentFactory`: This is the component factory, which will be used to create a new component. • `Index`: This is the optional parameter to specify the position in the view container at which the created component should be inserted. If this parameter is not specified, the component will be inserted at the last position in the view container. • `Injector`: This is an optional parameter that allows you to specify a custom injector for the created component. This allows you to provide additional dependencies for the created

	component. • `projectableNodes`: This is an optional parameter to specify nodes for content projection. This method returns a promise that is resolved when the instantiated component is compiled. The `Promise` resolves to a `ComponentRef` object, which can also be used to destroy the component again later on.

Tip

By default, a component created with the `ViewContainerRef.createComponent` method will inherit the injector from the parent component, which makes this process context aware. However, the `injector` parameter of the `createComponent` method is especially useful when you want to provide additional dependencies into the component that are not present on any parent injector.

Let's go back to our `PluginSlot` directive that is responsible for the instantiation of relevant plugin components.

First, let's think about the high-level requirements of our `PluginSlot` directive before we dive into the code:

- The plugin slot should contain a name input property so that this name can be referenced from plugins that want to provide components for the slot.
- The directive needs to react on changes of the `PluginService` and re-evaluate what plugin components need to be placed.
- In the initialization of the plugin slot, we need to obtain a list of the `PluginData` objects that are relevant to this particular slot. We should consult the `getPluginData` method of `PluginService` in order to get this list.
- Using the obtained list of the relevant `PluginData` objects, we'll be able to instantiate components that are associated with the placement information using the `ViewContainerRef` object of our directive.

Let's create our `PluginSlot` directive on the `lib/plugin/plugin-slot.js` path:

```
import {Directive, Input, Inject, provide, ViewContainerRef,
ComponentResolver, ReflectiveInjector} from '@angular/core';
import {PluginData} from './plugin';
import {PluginService} from './plugin-service';
@Directive({
  selector: 'ngc-plugin-slot'
})
export class PluginSlot {
  @Input() name;
  ...
}
```

The `name` input in our directive is very important for our plugin mechanism. By providing a name to the directive, we can define named extension points in our UI and later use this name in the `PluginPlacement` data of the plugin configurations:

```
  constructor(@Inject(ViewContainerRef) viewContainerRef,
              @Inject(ComponentResolver) componentResolver,
              @Inject(PluginService) pluginService) {
    this.viewContainerRef = viewContainerRef;
    this.componentResolver = componentResolver;
    this.pluginService = pluginService;
    this.componentRefs = [];
    // Subscribing to changes on the plugin service and re-
    // initialize slot if needed
    this.pluginChangeSubscription =
      this.pluginService
        .change.subscribe(() => this.initialize());
  }
```

In the constructor, we first inject the `ViewContainerRef` object, which is a reference to the view container of the directive. As we want to use the view container of the directive directly, there's no need to use `@ViewChild` here. If we want the view container of the current directive, we can simply use injection. We'll use this reference while we're instantiating components using the `ViewContainerRef.createComponent` method.

In order to resolve components and their factory, we inject the `ComponentResolver` instance.

The `PluginService` is injected for two reasons. First, we'd like to subscribe to any changes on the list of active plugins, and secondly, we use it to obtain relevant `PluginData` objects for this slot.

We use the `componentRefs` member to keep track of already instantiated plugin components. This will help us destroy them later on when a plugin gets deactivated.

Finally, we create a new subscription to `PluginService` and store the subscription into the `pluginChangeSubscription` member field. On any changes of the activated plugin list, we execute the `initialize` method on our component:

```
initialize() {
  if (this.componentRefs.length > 0) {
    this.componentRefs.forEach(
      (componentRef) => componentRef.destroy());
    this.componentRefs = [];
  }

  const pluginData =
    this.pluginService.getPluginData(this.name);

  pluginData.sort(
    (a, b) => a.placement.priority < b.placement.priority ?
       1 : a.placement.priority > b.placement.priority ? -1 : 0);
```

```
  return Promise.all(
    pluginData.map((pluginData) =>
      this.instantiatePluginComponent(pluginData))
  );
}
```

Let's look at the four parts of the initialize method in detail:

- First, we check whether this plugin slot already contains instantiated plugin components in the `componentRefs` member. If this is the case, we use the detach method of the `ComponentRef` objects to remove all existing instances. After this, we initialize the `componentRefs` member with an empty array.
- We use the `getPluginData` method of `PluginService` to obtain a list of the `PluginData` objects that are relevant for this particular slot. We pass the name of this slot to the method, so the `PluginService` will already provide us with a filtered list of plugin components that are interested to be placed in our slot.
- As there could be many plugins queueing for placement in our slot, we are using the priority property of the `PluginPlacement` objects to sort the list of the `PluginData` objects. This will ensure that plugin components with higher priority will be placed before the ones with a lower priority. This is a nice extra feature that will come in handy when we deal with a lot of plugins fighting for space.
- The last code part in our `initialize` method calls the `instantiatePluginComponent` method for each `PluginData` object in our list.

Now, let's create the `instantiatePluginComponent` method, which is called as a last step in the `initialize` method:

```
instantiatePluginComponent(pluginData) {
  return this.componentResolver
    .resolveComponent(pluginData.placement.component)
    .then((componentFactory) => {
      // Get the injector of the plugin slot parent component
      const contextInjector = this.viewContainerRef.parentInjector;
      // Preparing additional PluginData provider for the created
      // plugin component
      const providers = [
        provide(PluginData, {
            useValue: pluginData
        })
      ];
      // We're creating a new child injector and provide the
      // PluginData provider
      const childInjector = ReflectiveInjector
        .resolveAndCreate(providers, contextInjector);
      // Now we can create a new component using the plugin slot view
      // container and the resolved component factory
      const componentRef = this.viewContainerRef
        .createComponent(componentFactory,
                         this.viewContainerRef.length,
                         childInjector);
      this.componentRefs.push(componentRef);
    });
```

```
}
```

This method is responsible for the instantiation of an individual plugin component. Now, we can use the knowledge that we gained in this topic about the `ViewContainerRef.createComponent` method and the `ComponentResolver` object to instantiate components dynamically.

In addition to the inherited providers from the component where this plugin slot is placed, we'd like to provide `PluginData` to the injector of the instantiated plugin component. Using Angular's `provide` function, we can specify `pluginData` to resolve for any injection on the `PluginData` type.

The `ReflectiveInjector` class provides us with some static methods that are used to create injectors. We can use the `parentInjector` member on our view container to obtain the injector that is present in the plugin slot context. Then, we use the static `resolveAndCreate` method on the `ReflectiveInjector` class in order to create a new child injector.

In the first parameter of the `resolveAndCreate` method, we can provide a list of providers. Those providers will be resolved and made available in our new child injector. The second parameter of the `resolveAndCreate` method accepts the parent injector of the newly-created child injector.

Finally, we use the `createComponent` method of the `ViewContainerRef` object to instantiate the plugin component. As a second parameter to the `createComponent` method call, we need to pass the position in the view container. Here, we make use of the `length` property of our view container in order to place it at the very end. In the third parameter, we override the default injector of the component with our custom child injector. On success, we add the created `ComponentRef` object to our list of instantiated components.

Finalizing our plugin architecture

Congratulations, you've just built your own plugin architecture using Angular! We created a plugin API that can be used to create new plugins using the `PluginConfig` decorator. `PluginService` manages the whole plugin loading and provides the `PluginData` objects to the slots in our application using custom injectors. The `PluginSlot` directive can be used in the task management application to mark extension points in the user interface. Using the inheriting nature of the dependency injection in Angular, plugin components will be able to access whatever they require from their environment.

In the next section, we will create our first plugin using the plugin architecture that we just created.

Building an Agile plugin

In the previous section, we created a simple but effective plugin architecture, and we will now use this plugin API to build our first plugin in the task management application.

Before we get into the plugin details, we should first agree on where to make our application extensible. Our plugin system is based on the `PluginSlot` directives, which should be placed somewhere in our component tree so that plugins can expose components to these slots. For now, we decide to make two spots in our application extensible:

- `TaskInfo`: In the list of tasks displayed in a project, we currently render `Task` components. Besides the title of the task, the `Task` component displays additional information such as the task number, the date of creation, and milestones, as well as efforts information where applicable. This additional information is rendered on the `Task` component using the `TaskInfos` subcomponent. This is a good spot to provide extensibility for plugins so that they can add additional task information, which will be displayed on the task list overview.
- `TaskDetail`: Another great spot to provide extensibility is the `ProjectTaskDetails` component. This is where we can edit the details of a task, which makes it a great component to open up for extension by plugins.

Besides adding the `PluginSlot` directive to the directives list of the `TaskInfos` component, we modify the template located at `lib/task-list/task/task-infos/task-infos.html`:

```
...
<ngc-task-info title="Efforts"
               [info]="task.efforts | formatEfforts">
</ngc-task-info>
<ngc-plugin-slot name="task-info"></ngc-plugin-slot>
```

After including the `PluginSlot` directive and by setting the name input property to `task-info`, we provide an extension point for plugins where they can provide additional components.

Let's apply the same changes to the `ProjectTaskDetails` component template in `lib/project/project-task-details/project-task-details.html`:

```
...
<div class="task-details__content">
  ...
  <ngc-plugin-slot name="task-detail"></ngc-plugin-slot>
</div>
```

Right before the end of the task details content element, we include another plugin slot with the name `task-detail`. By providing components for this slot, plugins can hook into the edit view of tasks.

Okay, so our extension points are set up for plugins to provide additional components on a

task level. You can see that preparing these spots using the `PluginSlot` directive is really a piece of cake.

Now, we can look into the implementation of our Agile plugin, which will make use of the extension points that we just exposed.

The agile plugin that we will create will provide functionality to log story points on tasks. Story points are commonly used in Agile project management. They should provide a sense for complexity, and they are relative to a so-called reference story. If you want to know more about Agile project management and how to estimate using story points, I really recommend the book, *Agile Estimating and Planning* by Mike Cohn.

Let's start with our plugin class and the necessary configuration. We create the plugin outside our regular `lib` folder, just to indicate the portable nature of plugins.

We create a new `AgilePlugin` class on the `plugins/agile/agile.js` path:

```
import {PluginConfig, PluginPlacement} from '../../lib/plugin/plugin';

@PluginConfig({
  name: 'agile',
  description: 'Agile development plugin to manage story points on tasks',
  placements: []
})
export default class AgilePlugin {
  constructor() {
    this.storyPoints = [0.5, 1, 2, 3, 5, 8, 13, 21];
  }
}
```

The plugin class forms the central entry point of our plugin. We use the `PluginConfig` decorator, which we created as part of our plugin API. Besides the name and description, we also need to configure any placements where we map plugin components to application plugin slots. However, as we haven't got any plugin component yet to expose, our list remains empty for the moment.

It's also important to note that a plugin module always needs to default export the plugin class. This is just how we've implemented the plugin-loading mechanisms in our `PluginService` class.

Looking back at these two lines in the `loadPlugin` method of `PluginService` shows you that we rely on the default export of plugin modules:

```
return System.import(url).then((pluginModule) => {
    const Plugin = pluginModule.default;
...
```

When the plugin module is successfully loaded, we obtain the default export by referencing the `default` property on the module.

So far, we created our plugin entry module. This acts as a plugin configuration container, and it is not related to Angular in any way. Using the placements configuration, we can then expose our plugin Angular components once we've created them.

Agile task info component

Let's move on to the first Agile plugin component that we want to expose. First, we create the component, which will be exposed into the slot with the name `task-info`. Below the task title on the task list, our Agile information component should display the stored story points.

We create a new `Component` class on the `plugins/agile/agile-task-info/agile-task-info.js` path:

```
...
import {Task} from '../../../lib/task-list/task/task';

@Component({
  selector: 'ngc-agile-task-info',
  encapsulation: ViewEncapsulation.None,
  template,
  host: {
    class: 'task-infos__info'
  }
})
export class AgileTaskInfo {
  constructor(@Inject(Task) taskComponent) {
    this.task = taskComponent.task;
  }
}
```

You can see that we implemented a regular component here. There's nothing special about this component at all.

We import the `Task` component to get the type information to inject it in our constructor. As the plugin slot is placed inside of the `TaskInfos` component, which, in fact, is always a child of a `Task` component, this is a safe injection.

In the constructor, we then take the injected `Task` component and extract the task data into a local `task` member.

We also borrow the `task-infos__info` class of the `TaskInfos` component in order to get the same look like other task information that is already present on the task.

Let's take a look at the template of the `AgileTaskInfo` component that is located in the same path in the `agile-task-info.html` file:

```
<div *ngIf="task.storyPoints || task.storyPoints === 0">
  <strong>Story Points: </strong>{{task.storyPoints}}
</div>
```

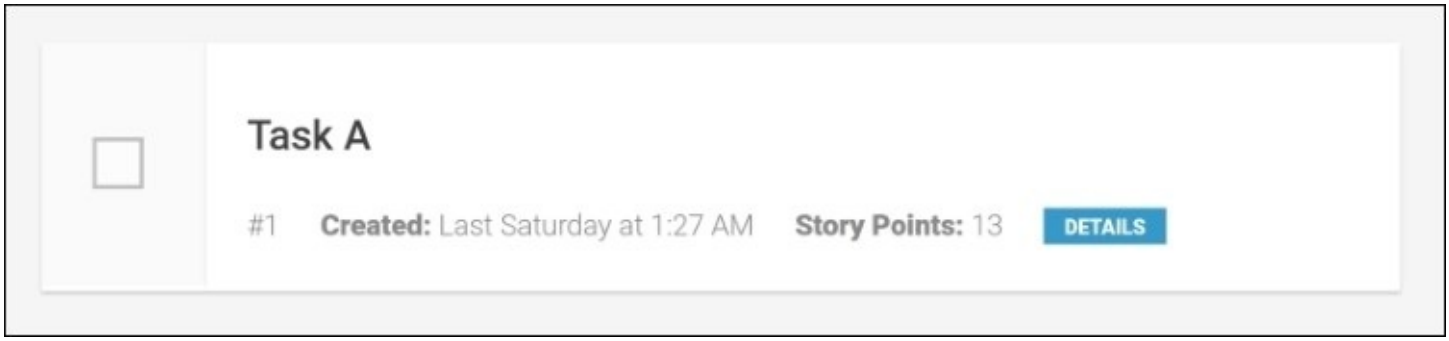
Following the same mark-up that we used in the `TaskInfo` component, we display the `storyPoints` tasks if present.

Alright, now we can expose the plugin component in the plugin configuration using a

`PluginPlacement` object. Let's make the necessary modification to our `agile.js` module file:

```
...
import {AgileTaskInfo} from './agile-task-info/agile-task-info';

@PluginConfig({
  name: 'agile',
  description: 'Agile development plugin to manage story points on tasks',
  placements: [
    new PluginPlacement({slot: 'task-info', priority: 1,
                         component: AgileTaskInfo})
  ]
})
export default class AgilePlugin {
  ...
}
```

Now, we include a new `PluginPlacement` object in our plugin configuration, which maps our `AgileTaskInfo` component to be exposed into the application plugin slot with the name `task-info`:



Task info displaying additional information that is provided by our Agile plugin

This would already be enough for the plugin to work. However, as we don't have any data filled as `storyPoints` on our tasks, this plugin wouldn't really show us anything at the moment.

Agile task details component

Let's create another plugin component, which can be used to enter story points. For this, we will create a new `AgileTaskDetail` component on the `plugins/agile/agile-task-detail/agile-task-detail.js` path:

```
...
import {Project} from '../../../lib/project/project';
import {ProjectTaskDetails} from '../../../lib/project/project-task-details/project-task-details';
import {Editor} from '../../../lib/ui/editor/editor';

@Component({
  selector: 'ngc-agile-task-detail',
  encapsulation: ViewEncapsulation.None,
  template,
  host: {class: 'agile-task-detail'},
  directives: [Editor]
})
export class AgileTaskDetail {
  constructor(@Inject(Project) project,
              @Inject(ProjectTaskDetails) projectTaskDetails) {
    this.project = project;
    this.projectTaskDetails = projectTaskDetails;
    this.plugin = placementData.plugin.instance;
  }

  onStoryPointsSaved(storyPoints) {
    this.projectTaskDetails.task.storyPoints = +storyPoints || 0;
    this.project.document.persist();
  }
}
```

There's nothing really fancy with this component either. Our target slot is the `task-detail` plugin slot, which is placed inside the `ProjectTaskDetails` component. Therefore, it's safe for both the `ProjectTaskDetails` and `Project` components to be injected into our plugin component. The `ProjectTaskDetails` component is used to obtain the task data in context. We use `LiveDocument` that is stored on the `Project` component to persist any changes that we make to the task data of the project.

We reuse an `Editor` component to obtain user input and store the input data in the `onStoryPointsSaved` call-back. This is the same mechanism we know from other areas where we use the `Editor` component. When the story points get edited, we first update the task data model that is stored in the `ProjectTaskDetails` component. After this, we can use the `LiveDocument` `persist` method to save the changes.

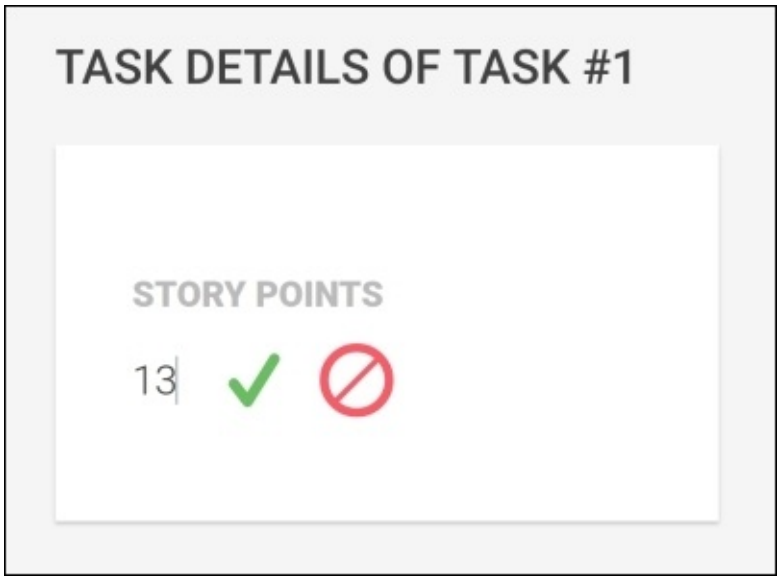
Let's look at the template of our `AgileTaskDetail` component in the `plugins/agile/agile-task-detail/agile-task-detail.html` file:

```
<div class="task-details__label">Story Points</div>
<ngc-editor [content]="projectTaskDetails.task?.storyPoints"
```

```
            [showControls]="true"
            (editSaved)="onStoryPointsSaved($event)"></ngc-editor>
```

We create a direct binding from the `content` input property of our editor to the `storyPoints` property of the task data.

When an edit is saved, we call the `onStoryPointsSaved` callback with the updated value:



Task details displaying the new Agile story points that are exposed by our Agile plugin

Before we expose our newly-created plugin component using a new `PluginPlacement` object on the plugin configuration, we'll enhance the component one more time. This would be nice if we provide two buttons on the component that allows the user to increase or decrease the story points to the next common story point value in range. As we already stored the list of common story points on the `Agile` plugin class, let's see how we can make use of this:

```
...
import {PluginData} from '../../../lib/plugin/plugin';

@Component({
  selector: 'ngc-agile-task-detail',
  ...
})
export class AgileTaskDetail {
  constructor(..., @Inject(PluginData) pluginData) {
    ...
    this.plugin = pluginData.plugin.instance;
  }
  ...
  increaseStoryPoints() {
   const current = this.projectTaskDetails.task.storyPoints || 0;
    const storyPoints = this.plugin.storyPoints.slice().sort((a, b) => a > b ? 1
: a < b ? -1 : 0);
```

```
    this.projectTaskDetails.task.storyPoints =
      storyPoints.find((storyPoints) => storyPoints > current) || current;
    this.project.document.persist();
  }

  decreaseStoryPoints() {
   const current = this.projectTaskDetails.task.storyPoints || 0;
    const storyPoints = this.plugin.storyPoints.slice().sort((a, b) => a < b ? 1
: a > b ? -1 : 0);
    this.projectTaskDetails.task.storyPoints =
      storyPoints.find((storyPoints) => storyPoints < current) || current;
    this.project.document.persist();
  }
}
```

While we previously injected the `Project` and `ProjectTaskDetails` components that are provided by the component level injectors, we now make use of the providers that we added during instantiation in our `PluginSlot` directive. Here, we provided `PluginData`, which we can now use to get a reference back to the plugin component.

The next higher or lower story point value is found by `increaseStoryPoints` and `decreaseStoryPoints`. This is done by searching the list of common story points that are stored on our `AgilePlugin` class. Using the plugin class instance that is present on the injected `PluginData`, we can easily access this list. After storing the modified story points, we then use the `LiveDocument` instance of the project component to persist the adjusted story points.

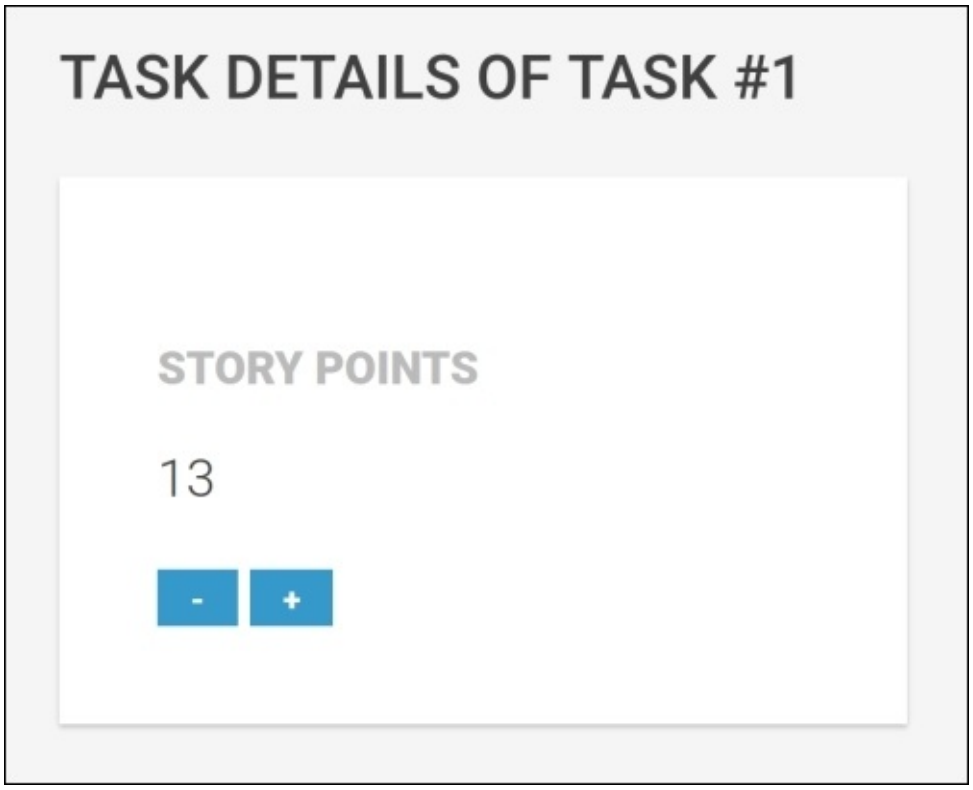
In the template of our `AgileTaskDetail` component, we simply add two buttons that allow the user to increase or decrease the story points that are based on our newly-created methods:

```
...
<button (click)="decreaseStoryPoints()"
        class="button button--small">-</button>
<button (click)="increaseStoryPoints()"
        class="button button--small">+</button>
```

Okay, let's now add the `AgileTaskDetail` component to the plugin configuration using a new `PluginPlacement` object, which references the `task-detail` plugin slot:

```
...
import {AgileTaskDetail} from './agile-task-detail/agile-task-detail';

@PluginConfig({
  ...
  placements: [
    new PluginPlacement({slot: 'task-info', priority: 1,
                         component: AgileTaskInfo}),
    new PluginPlacement({slot: 'task-detail', priority: 1,
                         component: AgileTaskDetail})
  ]
})
export default class AgilePlugin {
  ...
}
```

Isn't this great? You created a fully-portable plugin that enables the management of Agile story points on tasks.



The task details view with story points and additional increase/decrease buttons

The only thing that is left is to add the plugin to the list of plugins that should be loaded initially by the `PluginService` directive. For this, we'll create a `plugins.js` file on the root of our application and add the following content:

```
export default [
  '/plugins/agile/agile.js'
];
```

Now, if we launch our application, the plugin will be loaded by the `PluginService` and `PluginSlot` directives will instantiate the Agile plugin components where appropriate.

Recapitulating on our first plugin

Well done! You successfully implemented your first plugin! In this section, we used the API of our plugin architecture to create a plugin to manage Agile story points. We used the `PluginPlacement` class to map our plugin components to different slots in the UI of our application. We also made use of the `PluginData` object that is provided to each instantiated component in the plugin slot in order to access the plugin instance.

The advantage of implementing functionality like this inside a plugin should be obvious. We added an additional feature to our application without building up additional dependencies. Our Agile feature is completely portable. Third-party developers can write independent plugins, and they can be loaded by our system. This is a big advantage, and it helps us keep our core slim while providing great extensibility.

Managing plugins

We already built the core of the plugin architecture and a first plugin that runs in this system. We can use the `plugins.js` file on the root of our application to register plugins. The system is actually fully functional already. However, it would be nice to provide a way to manage our plugins during runtime.

In this section, we will build a new routable component, which will list all active plugins in the system. After we've done this, we'll also add some elements which allow users to unload active plugins as well as load new plugins during runtime. Due to the reactive nature of our plugin system, the browser does not need to be refreshed in order for newly-loaded plugins to become active. The moment a plugin is loaded, it will immediately be made available to relevant plugin slots.

Let's start with a new `ManagePlugins` component class on the `lib/manage-plugins/manage-plugins.js` path:

```
...
import {PluginService} from '../plugin/plugin-service';

@Component({
  selector: 'ngc-manage-plugins',
  ...
})
export class ManagePlugins {
  constructor(@Inject(PluginService) pluginService) {
    this.plugins = pluginService.change;
  }
}
```

Our `ManagePlugins` component is quite simple. We inject the `PluginService` into the constructor of the component and set the member field plugins to point to the change observable of `PluginService`. As we'll always get the latest plugin list emitted by this observable, we can then simply use the `async` pipe in the view to subscribe to the observable.

Let's look at the template of our new component in `lib/manage-plugins/manage-plugins.html`:

```
<div class="manage-plugins__l-header">
  <h2 class="manage-plugins__title">Manage Plugins</h2>
</div>
<div class="manage-plugins__l-main">
  <h3 class="manage-plugins__sub-title">Active Plugins</h3>
  <div class="manage-plugins__section">
    <table class="manage-plugins__table">
      <thead>
      <tr>
        <th>Name</th>
        <th>Url</th>
        <th>Description</th>
```

```
        <th>Placements</th>
      </tr>
      </thead>
      <tbody>
      <tr *ngFor="let plugin of plugins | async">
        <td>{{plugin.config.name}}</td>
        <td>{{plugin.url}}</td>
        <td>{{plugin.config.description}}</td>
        <td>
          <div *ngFor="let placement of plugin.config.placements"
               class="manage-plugins__placement">
            {{placement.slot}}
          </div>
        </td>
      </tr>
      </tbody>
    </table>
  </div>
</div>
```

We use an HTML table to display the list of active plugins. On the table body rows, we use the `NgFor` directive to iterate over the list of active plugins, which we subscribe to using the `async` pipe.

In the plugin objects, we got everything worth displaying already present. By iterating over the `PluginPlacement` objects that are stored on the `config` property on the plugin data, we can even display the slot names where our plugins provide components.

Now, the only thing left to do to enable our new component is to make it routable and to add it to the navigation of our application. Let's make the necessary modification in the `lib/app.js` module for this:

```
...
import {ManagePlugins} from './manage-plugins/manage-plugins';

@Component({
  selector: 'ngc-app',
  ...
})
@Routes([
  ...
  new Route({path: 'plugins', component: ManagePlugins})
])
export class App {
  ...
}
```

We added a new route, so let's add it to our navigation in `lib/app.html`:

```
<div class="app">
  <div class="app__l-side">
    <ngc-navigation [openTasksCount]="openTaskCount">
      ...
```

```
      <ngc-navigation-section title="Admin">
        <ngc-navigation-item title="Manage Plugins"
                             [link]="['/plugins']">
        </ngc-navigation-item>
      </ngc-navigation-section>
    </ngc-navigation>
  </div>
  <div class="app__l-main">
    <router-outlet></router-outlet>
  </div>
</div>
```

In a new `Admin` navigation section, we add a new `navigation-item` that links to the newly-created "plugins" route:

Our new ManagePlugins component displaying a table of active plugins and their exposed placements

Loading new plugins at runtime

We already have everything in place to provide a page to see all active plugins. However, we said that it would be nice to be able to manage this list. A user should be able to remove active plugins as well as manually load additional plugins.

Let's add these capabilities to our `ManagePlugins` component. Before we can do this, we'll need an additional method on our `PluginService` class, which is the part responsible for the loading of plugins. So far, we didn't consider the functionality to remove active plugins. Let's open `PluginService` in `lib/plugin/plugin-service.js` to add this functionality:

```
...
@Injectable()
export class PluginService {
  ...
  removePlugin(name) {
    const plugin = this.plugins.find(
      (plugin) => plugin.name === name);
    if (plugin) {
      const plugins = this.plugins.slice();
      plugins.splice(plugins.indexOf(plugin), 1);
      this.plugins = plugins;
      this.change.next(this.plugins);
    }
  }
}
```

Well, this was easy! We provide a new `removePlugin` method, which takes a plugin name as parameter. We then look up the plugin in the `plugins` array, and if a plugin was found with this name, we remove it from the list. Additionally, after we've removed the plugin, we emit a `change` event with the updated list. As all plugin slots in the application are subscribed to this change observable, they will update and reinitialize relevant plugin components automatically.

Let's now apply the necessary changes to our `ManagePlugins` component class in order to not only remove plugins but also to load additional plugins:

```
...
@Component({
  selector: 'ngc-manage-plugins',
  ...
})
export class ManagePlugins {
  constructor(@Inject(PluginService) pluginService) {
    ...
    this.pluginService = pluginService;
  }

  removePlugin(name) {
    this.pluginService.removePlugin(name);
  }
```

```
  loadPlugin(loadUrlInput) {
    this.pluginService.loadPlugin(loadUrlInput.value);
    loadUrlInput.value = '';
  }
}
```

Now, we also store the `PluginService` on our component. In the `removePlugin` and the `loadPlugin` functions, we delegate to the `PluginService` to take the necessary actions.

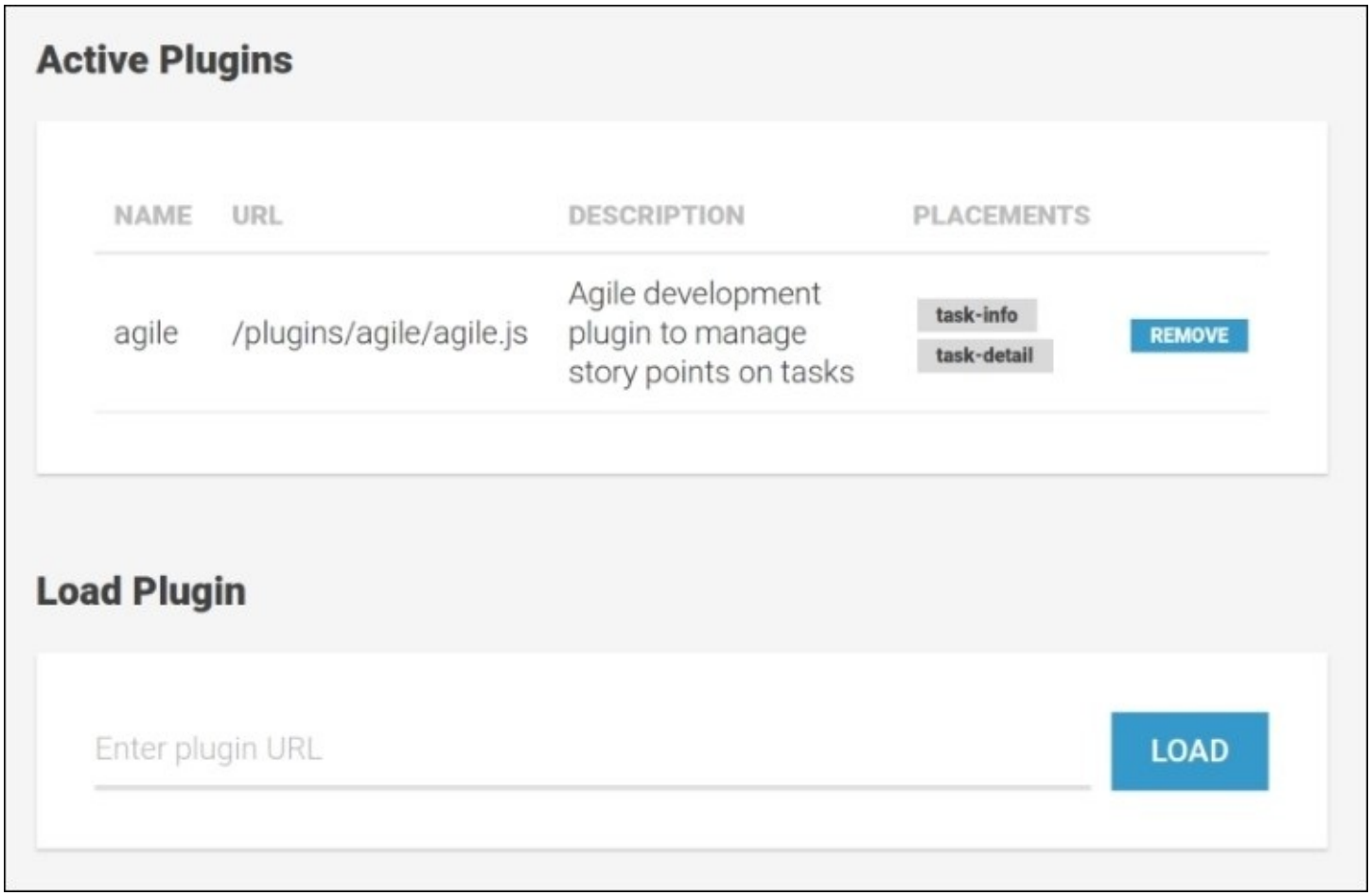
The `loadPlugin` method will receive an `ElementRef` object that points to the input field, where the user enters the URL from which we load a new plugin. We can pass the value of the input field to the `loadPlugin` method of `PluginService`, which deals with the rest. We also set the input field value to an empty string once we've submitted this call.

Let's open the template at `lib/manage-plugins/manage-plugins.html` to apply the required changes in the view of our component:

```
...
<div class="manage-plugins__l-main">
  <h3 class="manage-plugins__sub-title">Active Plugins</h3>
  <div class="manage-plugins__section">
    ...
        <td>
          <button (click)="removePlugin(plugin.name)"
                  class="button button--small">remove</button>
        </td>
    ...
  </div>
  <h3 class="manage-plugins__sub-title">Load Plugin</h3>
  <div class="manage-plugins__section">
    <div class="manage-plugins__load-elements">
      <input #loadUrlRef type="text"
             placeholder="Enter plugin URL"
             class="manage-plugins__load-url">
      <button (click)="loadPlugin(loadUrlRef)"
              class="button">Load</button>
    </div>
  </div>
</div>
```

We added an additional button for every listed plugin in the table, which contains a binding expression that calls the `removePlugin` method with the currently iterated plugin name.

We also added a new section after the listed plugins to load new plugins. In this section, we use an input field to enter the plugin URL as well as a button to execute the loading. Using a `loadUrlRef` local view reference, we can pass a reference to the input DOM element to the `loadPlugin` method on our component:



A completed ManagePlugins component with the ability to remove and load plugin modules at runtime

Now, we have everything in place to manage our plugins. Plugins initially loaded from URLs present in the root `plugins.js` file can now be unloaded using the REMOVE button in the plugins listing. New plugins can be loaded and activated by entering the URL of a plugin, which could be a local URL, bundled and mapped module, or even a remote URL on a different server.

Summary

In this chapter, we looked at different approaches on how to implement a plugin architecture. We then created our own design for a plugin architecture that leverages some Angular mechanisms and works on the concept of UI extension points that we call slots.

We implemented a plugin API that provides great developer experience by leveraging ES7 decorators to make the configuration of new plugins a piece of cake. We implemented the core of our plugin system using a service to load and unload plugins that are based on the SystemJS module loader. This allowed us to make use of the advanced loading possibilities that are provided by SystemJS. Plugins can be transpiled in real time, can be located on a local URL, remote URL, or even be bundled into the main application.

We implemented our first plugin, which provides some components to manage Agile story points on our tasks. The plugin was created outside our regular project `lib` folder, which should underline the portable nature of our plugin system.

Finally, we created a new routable component to manage plugins at runtime. Due to the reactive nature of our plugin system, plugins can be loaded and unloaded during application runtime without any unwanted side-effects.

When you're playing with the source code of this chapter, I highly recommend that you play with the loading mechanism of our plugin architecture. The flexibility that we achieved with very little effort is fantastic. You can unload the Agile plugin and load it again by providing the URL to the plugin main module. You can even try to place the whole plugins folder onto a remote server and load the plugin from there. Just make sure that you consider the necessary **Cross-Origin Resource Sharing** (**CORS**) headers.

The whole code for this chapter can be found in the ZIP file of the book resources that you can download from Packt Publishing. You can refer to the *Downloading the example code* section in the *Preface* of the book.

In the next and last chapter of this book, we'll look at how we can test the components that we've created so far. So, stay tuned for this overdue topic!

Chapter 11. Putting Things to the Test

Writing tests is crucial for the maintainability of your code. It's a known fact that having a good range of tests that cover most of your functionality is equally important as the functionality itself.

The first thing that comes to mind when thinking about tests is probably code quality assurance. You test the code that you write, so this is definitely true. However, there are many other important aspects of writing tests:

- **Resistance to unexpected change**: Your tests define what your code is supposed to do. They test whether your code conforms to your specifications. This has several benefits, where the most obvious is probably a resistance to unexpected change in the future. If you modify the code in the future, you'll less likely break your existing code because your tests will validate whether the existing functionality still works as specified.
- **Documentation**: Your tests define what your code should do. At the same time, they display the API calls that are required to use the concerned functionality. This is the perfect documentation for any developer. Whenever I want to understand how a library really works, the tests are the first thing that I look at.
- **Avoiding unnecessary code**: The practice of writing tests forces you to limit your code to fulfil the requirements of your specification and nothing more. Any code in your application that is not reached in your automated tests can be considered dead code. If you stick to a merciless refactoring approach, you'd then remove such unused code ASAP.

So far, we haven't considered testing in our book at all, and given its importance, you may wonder why I come up with this now in the last chapter. In a real project, we'd definitely create tests much earlier if not at first. However, I hope you understand that in this book, we postponed this rather important topic until the end. I really love testing, but as we're mainly focused on the component architecture of Angular, placing this chapter at the end seemed more logical.

In this chapter, we'll look into how to perform proper unit testing on your components. We'll focus on unit testing; automated end-to-end testing is beyond the scope of this book. Still, we'll look into how to test user interaction on components, although not on the level it would be done in end-to-end testing.

In this chapter, we will delve into the following topics:

- An introduction to the Jasmine testing framework
- Writing simple JavaScript tests for components
- Creating a `tests.html` file, which serves as an in-browser test runner
- Creating Jasmine spies and observing component output properties
- Learning about Angular testing utilities, such as `inject`, `async`, `TestComponentBuilder`, `DebugElement`, and more

- Mocking components
- Mocking existing services
- Creating tests for our `AutoComplete` UI component
- Creating tests for our plugin system

An introduction to Jasmine

Jasmine is a very simple testing framework, which comes with an API that allows you to write **Behavior-driven Development** (**BDD**) style tests. BDD is an agile software development process of defining specifications in a written format.

In BDD, we define that an agile user story consists of multiple scenarios. These scenarios closely relate to or even replace the acceptance criteria of a story. They define requirements on a higher level, and they are mostly written narrative. Each scenario then consists of three parts:

- **Given**: This part is used to describe the initial state of the scenario. The test code is where we perform all the setup that is needed to execute the test scenario.
- **When**: This part reflects the changes that we perform to the system under test. Usually, this part consists of some API calls and actions that reflect the behavior of a user of the system.
- **Then**: This part specifies what the system should look like after the given state and the changes applied in the *when* part. In our code, this is the part that is usually at the end of our tests function, where we use assertion libraries to verify the state of the system.
- Jasmine comes with an API that makes it very easy to write tests which structure according to the BDD style. Let's look at a very simple example of how we use Jasmine to write a test for a shopping cart system:

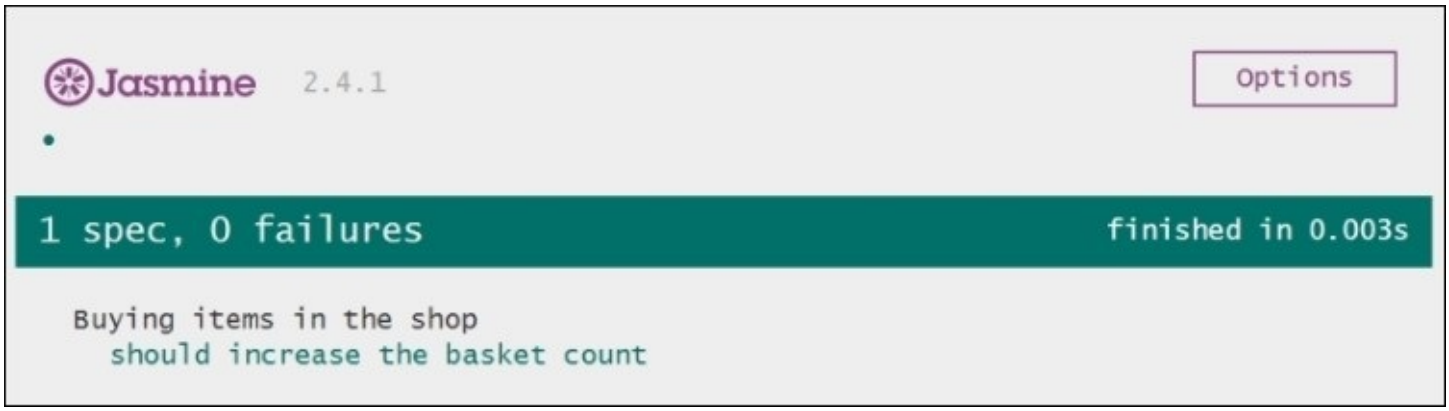
```
describe('Buying items in the shop', () => {
  it('should increase the basket count', () => {
    // Given
    const shop = new Shop();
    // When
    shop.buy('Toothpaste');
    shop.buy('Shampoo');
    // Then
    expect(shop.basket.length).toBe(2);
    expect(shop.basket).toContain('Toothpaste');
    expect(shop.basket).toContain('Shampoo');
  });
});
```

Jasmine provides us with a `describe` function, which allows us to group certain scenarios on the same subject. In this example, we used the `describe` function to register a new test suite for tests about buying items in a shop.

Using the `it` function, we can register individual scenarios, which we'd like to get tested. In the `describe` callback function, we can register as many scenarios using the `it` function as we like.

Inside the callback function of the Jasmine `it` function, we can start writing our test. We use a `BDD` style to structure the code inside our test.

You don't necessarily need to run Jasmine in the browser, but if you do this, you'll get a nice summary report of all tests and their state:



Jasmine provides a nice visual report over all your test specifications, which also allows you to rerun individual tests and provides you with more options

Jasmine comes in three parts that are relevant to us:

- **Jasmine core**: This contains the test definition APIs, the assertion library, and all the other core parts of the testing framework
- **Jasmine HTML**: This is the HTML reporter, which will write all tests results to the browser document and even provide options to rerun individual tests
- **Jasmine boot**: This is the file that bootstraps the Jasmine framework for the browser and performs any setup that is needed with the HTML reporter

In our project, we will use Jasmine and the preceding parts directly from a CDN, so we don't need to install anything to get started. We create a new `tests.html` file, which will serve as a runner for our tests. In conjunction with `live-server`, we can always have this page open in our browser. This way we'll get immediate feedback on our tests while developing.

Tip

Jasmine also plays nice with test runners such as Karma to run your tests. Karma is a popular test runner, which allows you to run your tests in parallel using the Karma CLI or integrate it in your build pipeline. This also allows you to run tests in different browsers. In this chapter, we will use the Jasmine HTML and Jasmine boot to run our tests directly in the browser. This allows us to skip the rather complex setup that we'd need to undertake if we used Karma as our test runner.

Let's look at the code of the `tests.html` file that we create in the root folder of our application, right next to the `index.html` file, which is already present:

```
...
<script src="https://cdnjs.cloudflare.com/ajax/libs/es6-shim/0.35.0/es6-
```

```
shim.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/2.0.0-
beta.17/angular2-polyfills.js"></script>
<script src="jspm_packages/system.js"></script>
<script src="config.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine-
html.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/boot.js">
</script>
```

Besides loading the usual suspects for our Angular application (ES6 shim, Angular polyfills, and SystemJS), we now also load the three main components of Jasmine.

By default, Jasmine executes all registered tests on the window's `load` event. However, as we will load our tests using SystemJS, we need to defer the bootstrap of Jasmine until SystemJS has completely loaded our tests:

```
<script>
  window._jasmineOnLoad = window.onload;
  window.onload = null;
    return System.import('./all.spec')
    .then(window._jasmineOnLoad)
    .catch(console.error.bind(console));
  ...
</script>
```

We first put aside the function that was registered by Jasmine boot on `window.onload`. We store the function in a temporary `_jasmineOnLoad` global variable.

Now, we use SystemJS to import our entry point module for our tests, which will be stored in the `all.spec.js` file. SystemJS returns a `Promise` that will be resolved if the test module has been loaded and executed successfully. We can use the `then` function of the returned `Promise` to execute the Jasmine boot function stored in `window._jasmineOnLoad`. In this way, we make sure that Jasmine is booted after all our tests have been registered.

Writing our first test

Now that we are all set with the Jasmine setup, we can start writing our first test. In this section, we will create a first test for the `AutoComplete` component that we created in Chapter 8, *Time Will Tell*, of this book.

As Angular components are just classes, we can already test a lot of the functionality by instantiating the `Component` class and testing its methods. Tests that can be performed like this should always be considered first. These tests can run without Angular bootstrapping the component.

The `AutoComplete` component filters displayed results based on the available items and a filter criteria. In the following test, we'll verify that the `filter` method on the component works as expected.

Tip

In this book, we follow the practice to store test files by appending a `.spec.js` file to the name of the file that has to be tested. We'll also store these test files in the same folder of the subject. This makes it much easier to keep the context.

We'll create a new `auto-complete.spec.js` file in the folder of the `AutoComplete` component at `lib/ui/auto-complete`:

```
import {describe, expect, it,} from '@angular/core/testing';
import {AutoComplete} from './auto-complete';

describe('AutoComplete', () => {
  it('should filter items correctly', () => {
    // Given
    const autoComplete = new AutoComplete();
    autoComplete.items = ['One', 'two', 'three'];
    // When
    autoComplete.filterItems('o');
    // Then
    expect(autoComplete.filteredItems).toEqual(['One', 'two']);
  });
});
```

As we loaded Jasmine prior to executing our test, we could rely on the global `describe`, `it`, and `expect` functions that are exposed by Jasmine. However, Angular provides us with some nice wrappers of the Jasmine functions, which we can import from the module located in `@angular/core/testing`.

As you can see, we don't really need to activate the `AutoComplete` component in order to test some of its functionality. By simply testing the component class, we can already execute some of our executable specifications.

We follow a BDD approach to structure our test, and in the `Given` section, we instantiate a new `AutoComplete` component class, and then we initialize the items list with some test items. Even if the items field is actually a component input, we can simply disregard this fact in order to test the filtering functionality.

In the `When` section of our test, we actually call the `filterItems` method of the component class and test whether it does filter the items according to the specification.

In the `Then` section, we use the `expect` function of Jasmine in order to assert the expected state after the `When` section. As the component should filter all items with partial and case-insensitive matches of the filter criteria, the expected value in `filteredItems` should be an array with the `One` and `two` items.

We use the assertion `toEqual` function in order to perform a deep equal check. If we use the `toBe` matcher, we'd compare the references of the two arrays, which will result in a negative match.

This is it for our first test. What's left to do still is to create our main test module that is loaded in the `tests.html` file.

We created the main entry point for all our tests in a `all.spec.js` file on the root path of our application. This file will then include all specification files that we create in our application:

```
import './lib/ui/auto-complete/auto-complete.spec';
```

This is currently all that we need to make our test run. We simply import the test file that we just created. Now, `tests.html` will use SystemJS to load our `all.spec.js` file, and here, we then load the `auto-complete-spec.js` file.

We can now start `live-server` in the root path of our application and navigate to `http://127.0 .0.1:8080/tests.html` in our browser. As `live-server` will reload our browser on changes, we can start adding new tests while we constantly get updates on our test state in the browser.

Spying on component outputs

A common practice in testing is to use spy function calls during the execution of tests and then evaluate these calls, checking whether all functions have been called correctly.

Jasmine provides us with some nice helpers in order to use spy function calls. We can use the `spyOn` function of Jasmine in order to replace the original function with a spy function. The spy function will record any calls, and we can later on evaluate how many times it was called and with what parameters.

Let's look at a simple example of how to use the `spyOn` function:

```
class Calculator {
  multiply(a, b) {
    return a * b;
  }

  pythagorean(a, b) {
    return Math.sqrt(this.multiply(a, a) + this.multiply(b, b));
  }
}
```

We will test a simple `Calculator` class that has two methods. The `multiply` method simply multiplies two numbers and returns the result. The `pythagorean` method calculates the hypotenuse of a right-angled triangle with two sides, `a` and `b`.

You might remember the formula for the Pythagorean theorem from your early school days:

$a^2 + b^2 = c^2$

We will use this formula to produce `c` from `a` and `b` by getting the square root of the result of `a*a + b*b`. For the multiplications, we'll use our `multiply` method instead of using arithmetic operators directly.

Now, we'd want to test our calculator `pythagorean` method, and as it uses the `multiply` method to multiply `a` and `b`, we can spy on this method to verify our test result in depth:

```
describe('Calculator pythagorean function', () => {
  it('should call multiply function correctly', () => {
    // Given
    const calc = new Calculator();
    spyOn(calc, 'multiply').and.callThrough();
    // When
    const result = calc.pythagorean(6, 8);
    // Then
    expect(result).toBe(10);
    expect(calc.mul).toHaveBeenCalled();
    expect(calc.mul.calls.count()).toBe(2);
    expect(calc.mul.calls.argsFor(0)).toEqual([6, 6]);
    expect(calc.mul.calls.argsFor(1)).toEqual([8, 8]);
```

```
  });
});
```

The `spyOn` function of Jasmine takes an object as first parameter and the function name on the object which we'd like to spy on.

This will effectively replace the original `multiply` function on our class instance with a new spy function of Jasmine. By default, spy functions will only record function calls, and they won't delegate the call further to the original function. We can use the `.and.callThrough()` function to specify that we'd like Jasmine to call the original function. This way our spy function will act as a proxy and record any calls at the same time.

In the `Then` section of our test, we can then inspect the spy function. Using the `toHaveBeenCalled` matcher, we can check whether the spy function was called after all.

Using the `calls` property of the spy function, we can inspect in more detail and verify the call count as well as the arguments that individual calls received.

Using the knowledge that we gained about Jasmine spies, we can now apply that to our component tests. As we know that all output properties of components contain an `EventEmitter`, we can actually spy on them to check whether our component sends output.

Inside components, we call the `next` method on `EventEmitter` in order to send output to parent component bindings. As this is an asynchronous operation and we'd also like to test our components without needing to involve parent components, we can simply spy on the `next` method of our output properties.

In the next two tests for our `AutoComplete` component, we'd like to verify the functionality when we save an edit in the `Editor` child component. Let's quickly recap on this behavior:

- On saved edits, we get the `onEditSaved` method on the `AutoComplete` component that is called
- If the saved value is an empty string, the `AutoComplete` component should emit a `selectedItemChange` event with a `null` value
- If the saved value is no empty string and the value is not present in the items of the `AutoComplete` component, an `itemCreated` event should be emitted

Let's create the tests for the previous expected behavior to the already existing `lib/ui/auto-complete/auto-complete.spec.js` test file:

```
  ...
  it('should emit selectedItemChange event with null on empty content being
saved', () => {
    // Given
    const autoComplete = new AutoComplete();
    autoComplete.items = ['one', 'two', 'three'];
    autoComplete.selectedItem = 'three';
    spyOn(autoComplete.selectedItemChange, 'next');
```

```
    spyOn(autoComplete.itemCreated, 'next');

    // When
    autoComplete.onEditSaved('');

    // Then
    expect(autoComplete.selectedItemChange.next).toHaveBeenCalledWith(null);
    expect(autoComplete.itemCreated.next).not.toHaveBeenCalled();
  });
```

We create two Jasmine spies here. The first one spies on the `selectedItemChange` output property, while the second one spies on the `itemCreated` output property.

After simulation, the editor was saved with an empty string. We can start verifying our spies in the `Then` section of our test.

The `next` function of the `selectedItemChange` event, `EventEmitter`, should have been called with a `null` value, while `next` of `itemCreated` shouldn't have been called at all. We can use the `not` property on the returned expectation object to invert the matcher.

Let's add a second test for the behavior when an editor was saved with a value that does not yet exist in the `AutoComplete` component:

```
  it('should emit an itemCreated event on content being saved which does not
match an existing item', () => {
    // Given
    const autoComplete = new AutoComplete();
    autoComplete.items = ['one', 'two', 'three'];
    autoComplete.selectedItem = 'three';
    spyOn(autoComplete.selectedItemChange, 'next');
    spyOn(autoComplete.itemCreated, 'next');

    // When
    autoComplete.onEditSaved('four');

    // Then
    expect(autoComplete.selectedItemChange.next).not.toHaveBeenCalled();
    expect(autoComplete.itemCreated.next).toHaveBeenCalledWith('four');
  });
```

This time, we simulate a saved edit with a value, which isn't an empty string and does not exist in the autocomplete items already.

In the `Then` section of our code, we evaluate the spies and expect that the `itemCreated.next` function was called with a `four` string.

Using Jasmine spies, we managed to test our component output successfully without the need to bootstrap Angular. We performed these tests solely on the component class and by creating spies on the `EventEmitter` that is present on all output properties.

Utilities to test components

So far, we tested our components with plain vanilla JavaScript. The fact that components are in just regular classes make this possible. However, this can only be done for very simple use-cases. As soon as we'd like to test components for things that involve template compilation, user interaction on components, change detection, or dependency injection, we'll need to get a little help from Angular to perform our tests.

Angular comes with a whole bunch of testing tools that help us out here. In fact, the platform-agnostic way that Angular is built allows us to exchange the regular view adapter with a debug view adapter. This enables us to render components in such a way that allows us to inspect them in great detail.

To enable the debugging capabilities of Angular while rendering components, we need to modify our main entry point for our tests first.

Let's open up `all.spec.js` to make the necessary modifications:

```
import {setBaseTestProviders} from '@angular/core/testing';
import {TEST_BROWSER_DYNAMIC_PLATFORM_PROVIDERS,
TEST_BROWSER_DYNAMIC_APPLICATION_PROVIDERS} from '@angular/platform-browser-
dynamic/testing';

setBaseTestProviders(TEST_BROWSER_PLATFORM_PROVIDERS,
TEST_BROWSER_APPLICATION_PROVIDERS);

import './lib/ui/auto-complete/auto-complete.spec';
import './lib/plugin/plugin.spec';
```

Using the `setBaseTestProviders` function of the `@angular/core/testing` module, we can actually initialize a test platform injector, which will then be used in the context of our Angular testing. This function takes two arguments where the first one is an array of platform providers, and the second one is an array of application providers.

From the `@angular/platform-browser-dynamic/testing` module, we can import two constants that contain an already prepared list for both platform and application-level dependencies. Here are some of the providers present in these constants:

- **Platform-level providers**: These consist mostly of platform initialization providers to debug
- **Application-level providers**: These consist of the following:
  - `DebugDomRootRenderer`: This overrides the default `DomRenderer` in the browser and enables debugging of elements using `DebugElement` and probing
  - `MockDirectiveResolver`: This overrides the default `DirectiveResolver` and allows overriding of directive metadata for testing purposes
  - `MockViewResolver`: This overrides the default `ViewResolver` and allows overriding of component view specific metadata

Using the `setBaseTestProviders` function and the imported constants with the debugging providers, we can now initialize our test environment. After calling this function and passing our providers, Angular is set up for testing.

Injecting in tests

Injecting Angular dependencies in tests is made easy by two helper functions that we can use. The `inject` and `async` functions are available through the `@angular/core/testing` package, and they help us inject dependencies in our tests.

Let's look at this simple example where we inject the document element using the `inject` wrapper function. This test is irrelevant for our application, but it illustrates how we can now make use of injection in our tests:

```
import {describe, expect, it, inject} from '@angular/core/testing';
import {DOCUMENT} from '@angular/platform-browser';

describe('Application initialized with test providers', () => {
  it('should inject document', inject([DOCUMENT], (document) => {
    expect(document).toBe(window.document);
  }));
});
```

We can simply use `inject` to wrap our test function. The `inject` function accepts an array as the first parameter that should include a list of injectables. The second parameter is our actual test function, which will now receive the injected document.

The `async` function on the other hand helps us with a different concern too. What if our tests actually involve asynchronous operations? Well, a standard asynchronous Jasmine test would look like the following:

```
describe('Async test', () => {
  it('should be completed by calling done', (done) => {
    setTimeout(() => {
      expect(true).toBe(true);
      done();
    }, 2000);
  });
});
```

Jasmine provides us with a nice way to specify asynchronous tests. We can simply use the first parameter of our test functions, which resolves to a callback function. By calling this callback function, in our case we named it `done`, we tell Jasmine that our asynchronous operations are done, and we would like to finish the test.

Using callbacks to indicate whether our asynchronous test is finished is a valid option. However, this can make our test quite complicated if many asynchronous operations are involved. It's sometimes even impossible to monitor all the asynchronous operations that are happening under the hood, which also makes it impossible for us to determine the end of our test.

This is where the `async` helper function comes into play. Angular uses a library called Zone.js to monitor any asynchronous operation in the browser. Simply put, Zone.js hooks into any

asynchronous operation and monitors where they are initiated as well as when they are finished. With this information, Angular knows exactly how many pending asynchronous operations there are.

If we're using the `async` helper, we tell Angular to automatically finish our test when all asynchronous operations in our test are done. The helper uses Zone.js to create a new zone and determine whether all the microtasks executed within this zone are finished.

Let's look at how we can combine injection with an asynchronous operation in our test:

```
import {describe, expect, it, inject, async} from '@angular/core/testing';
import {DOCUMENT} from '@angular/platform-browser';

describe('Application initialized with test providers', () => {
  it('should inject document', async(inject([DOCUMENT], (document) => {

      setTimeout(() => {
        expect(document).toBe(window.document);
      }, 2000);
    }))
  );
});
```

By combining `inject` with `async` (wrapping), we now have an asynchronous operation in our test. The `async` helper will make our test wait until all asynchronous operations are completed. We don't need to rely on a callback, and we have the guarantee that even internal asynchronous operations will complete before our test finishes.

Tip

Zone.js is designed to work with all asynchronous operations in the browser. It patches all core DOM APIs and makes sure that every operation goes through a zone. Angular also relies on Zone.js in order to initiate change detection.

Test component builder

Angular comes with another very important testing utility to test components and directives. So far, we only tested the component class of our components. However, as soon as we need to test components and their behavior in our application, this involves a few more things:

- **Testing the view of components**: It's sometimes required that we test the rendered view of components. With all the bindings in our view, dynamic instantiation using template directives and content insertion, it's required that we can have a way to test all this behavior.
- **Testing change detection**: As soon as we update our model in our component class, we want to test the updates that are performed via change detection. This involves the whole change detection behavior of our components.
- **User interaction**: Our component templates probably contain a set of event bindings, which trigger some behavior on user interaction. We'd also need a way to test the state after some user interaction.
- **Overriding and mocking**: In a testing scenario, it's sometimes required to mock certain areas in our components in order to create a proper isolation for our test. In unit testing, we should be concerned only about the specific behavior that we want to test.

The `TestComponentBuilder`, which is available through the `@angular/compiler/testing` package, helps us exactly with the previous concerns. It's our main tool to test components.

`TestComponentBuilder` is provided to the test application injector, which we initialized in our `all.spec.js` module using the `setBaseTestProviders` function. The reason for this is that the builder itself also relies on a lot of platform and application dependencies to create components. As all our dependencies now come from the test injector and most of them are overridden to enable inspection, this makes perfect sense.

Let's look at a very simple example of how we can use `TestComponentBuilder` to test the view rendering of a dummy component:

```
@Component({
  selector: 'dummy-component',
  template: 'dummy'
})
class DummyComponent {}

describe('Creating a component with TestComponentBuilder', () => {
  it('should render its view correctly', async(inject([TestComponentBuilder],
(tbc) => {
      tbc.createAsync(DummyComponent).then((fixture) => {
        // When
        fixture.detectChanges();
        // Then
        expect(fixture.nativeElement.textContent).toBe('dummy');
      });
    }))
  );
```

```
});
```

As `TestComponentBuilder` is exposed in the test injector, we need to use dependency injection to get hold of the instance. We use the `inject` helper for this purpose. As creating a component is an asynchronous operation, we also need to make our test wait for completion using the `async` helper.

In our test function, we call the `createAsync` method of `TestComponentBuilder` and pass a reference to `DummyComponent`, which we want to create. This method returns a `Promise`, which will resolve once the component is successfully compiled.

In the `then` callback of the returned promise, we'll receive a special fixture object of the `ComponentFixture` type. We can then call the `detectChanges` method on this fixture object, which will execute change detection on the created component. After this initial change detection, the view of our dummy component is updated. We can now use the `nativeElement` property of the fixture in order to access the root DOM element of the created component.

Let's look at the `ComponentFixture` type and the available fields in more detail:

Member	**Description**
`detectChanges()`	This executes change detection on the root component that was created in the context of the fixture. The template bindings will not be evaluated automatically after creating a component using `TestComponentBuilder`. It's our own responsibility to trigger change detection. Even after we change the state of our components, we'd need to trigger change detection again.
`destroy()`	This method destroys the underlying component and performs any cleanup that is required. This can be used to test the `OnDestroy` component's lifecycle.
`componentInstance`	This property points to the component class instance, and this is our main interaction point if we want to interact with the component.
`nativeElement`	This is a reference to the native DOM element at the root of the created component. This property can be used to inspect the rendered DOM of our component directly.
`elementRef`	This is the `ElementRef` wrapper around the root element of the created component.

`debugElement`	This property points to an instance of `DebugElement` that was created by `DebugDomRootRenderer` in the component view rendering pipeline. The debug element provides us with some nice utilities to inspect the rendered element tree and testing user interaction. We'll take a closer look at this later in another section.

We've now looked at a very simple dummy component and how to test it using `TestComponentBuilder` in conjunction with the `inject` and `async` helper functions.

This is great, but it doesn't really reflect the complexity that we face when we need to test real components. Real components have a lot more dependencies than our dummy component. We rely on child directives and probably on injected services to obtain data.

Of course, the `TestComponentBuilder` also provides us with the tools that we need in order to test more complex components and keep the necessary isolation in a unit test.

Let's first look at an example where we'd like to test a `ParentComponent` component, which uses a `ChildComponent` component to render a list of numbers. As we'd only like to test `ParentComponent`, we're not interested in how `ChildComponent` renders this list. We want to remove the behavior of the child component from our test by providing a mock component for `ChildComponent` during our test, which allows us to easily verify that the data is received by the child component:

```
@Component({
  selector: 'child',
  template:'<ul><li *ngFor="let n of numbers">Item: {{n}}</li></ul>'
})
class ChildComponent {
  @Input() numbers;
}

@Component({
  selector: 'parent',
  template: '<child [numbers]="numbers"></child>',
  directives: [ChildComponent]
})
class ParentComponent {
  numbers = [1, 2, 3];
}
```

This is our starting point. We have two components, where we'll only be interested in testing the parent component. However, the child component is required by the parent component, and it implies a very specific way to render the numbers that are passed by the parent. We would only like to test whether our numbers were passed successfully to the child component. We don't want to involve the rendering logic of the child component in our test. This is very important because changing only the child component could then break our parent component

test, which we want to avoid.

The thing we want to achieve now is to create a mock of our child component in the context of our test:

```
@Component({
  selector: 'child',
  template: '{{numbers.toString()}}'
})
class MockChildComponent {
  @Input() numbers;
}
```

In our `MockChildComponent` class, it's important that we use the same selector property as the real component. Otherwise, the mocking will not work. In the template, we use a very simple output of the numbers input, which enables an easy inspection.

It's also important that we provide the same input properties as the original component. Otherwise, we won't imitate the real component correctly.

Now, we can go ahead and perform our test. Using an additional method of `TestComponentBuilder`, we are able to override the real `ChildComponent` with our mock component:

```
describe('ParentComponent', () => {
  it('should pass data to child correctly', async(inject([TestComponentBuilder],
(tbc) => {
      tbc
       .overrideDirective(ParentComponent, ChildComponent, MockChildComponent)
       .createAsync(ParentComponent).then((fixture) => {
         fixture.detectChanges();
         expect(fixture.nativeElement.textContent).toBe('1,2,3');
       });
    }))
  );
});
```

Using the `overrideDirective` method on `TestBuilderComponent`, we can modify the parent component's `directives` metadata before we create it. In this way, we're able to exchange the real child component with our `MockChildComponent` class.

As a result, we decouple `ParentComponent` from `ChildComponent` in the context of our test. We need this level of separation in order to create a proper isolation of our unit test. As our mock child component simply renders the string representation of the passed array, we can easily test the text content of our fixture.

Tip

The definition of a unit test is to test a single unit and isolate the unit from any dependencies. If we want to stick to this paradigm, we'd need to create a mock for every dependent

component. This can easily get us into a situation where we need to maintain more complexity only for the sake of our tests. The key here lies in finding the right balance. You should mock dependencies that have a great impact on our subject and ignore dependencies that have low impact on the functionality we'd like to test.

Let's look at a different use case where we have a component that injects a service in order to obtain data. As we also want to test only our component and not the service it relies on, we somehow need to sneak in a mock service instead of the real service into our component. `TestComponentBuilder` also provides a method to modify the `providers` metadata of directives, which comes in very handy for this case.

First, we declare our base component and a service that it relies on. In this example, the `NumbersComponent` class injects the `NumbersService` class, and it obtains an array with numbers from it:

```
@Injectable()
class NumbersService {
  numbers = [1, 2, 3, 4, 5, 6];
}

@Component({
  selector: 'numbers-component',
  template: '{{numbers.toString()}}',
  providers: [NumbersService]
})
class NumbersComponent {
  constructor(@Inject(NumbersService) numbersService) {
    this.numbers = numbersService.numbers;
  }
}
```

Now, we need to create a mock service that provides the data required in our test and isolates our component from the original service:

```
@Injectable()
class MockNumbersService extends NumbersService {
  numbers = [1, 2, 3];
}
```

In this simplified example, we just provide a different set of numbers. However, in a real mocking case, we can exclude a lot of steps that are unnecessary and could potentially create side effects. Using a mock service also ensures that our test, which is focused on the `NumbersComponent` class, will not break because of a change in the `NumbersService` class.

By extending the real service, we can leverage some of the behavior of our original service while overriding certain functionality in our mock. You need to be careful with this approach though, as we rely on the original service by doing this. If you'd like to create a fully isolated test, you should probably override all methods and properties. Or you can create a completely independent mock service, which provides the same methods and properties that are used in

your test.

Tip

When using TypeScript, you should use interfaces for this purpose where both your real service as well as your mock service implement the same interface.

Let's now look at the test case and how we can use `TestComponentBuilder` to provide our mock service instead of the real one:

```
describe('NumbersComponent', () => {
  it('should render numbers correctly', async(inject([TestComponentBuilder],
(tbc) => {
      tbc
       .overrideProviders(NumbersComponent, [
         provide(NumbersService, {
           useClass: MockNumbersService
         })
       ])
       .createAsync(NumbersComponent).then((fixture) => {
         fixture.detectChanges();
         expect(fixture.nativeElement.textContent).toBe('1,2,3');
       });
    }))
  );
});
```

Using the `overrideProviders` method on `TestComponentBuilder`, we can provide additional providers to the component under test. This allows us to override existing providers that are already present on the component. Using the `provide` function of the `@angular/core` module, we can create a provider which provides on requests for `NumberService` but also resolves to a `MockNumberService`.

`TestComponentBuilder` allows us to perform tests in a very simple, isolated, and flexible fashion. It plays a major role when writing unit tests for components. If you'd like to read more about the available methods on `TestComponentBuilder`, you can visit the official documentation website at https://angular.io/docs/ts/latest/api/core/testing/TestComponentBuilder-class.html.

Now, it's time to use what we learned about `TestComponentBuilder` service and start to test our application components in action!

Testing components in action

In the previous topic, we learned about the `TestComponentBuilder` service and how to use it to create components in our testing environment. We learned about the `inject` and `async` helpers as well as how to mock components and services.

Let's now use this knowledge to work on our tests for the `AutoComplete` component. Let's add another test to the `auto-complete.spec.js` file on the `lib/ui/auto-complete` path.

As the `AutoComplete` component relies on the rather complex `Editor` component, it's probably a good idea to mock our `Editor` component before we start writing a test:

```
@Component({
  selector: 'ngc-editor',
  template: '{{content}}'
})
export class MockEditor {
  @Input() content;
}
```

This might look a bit tenuous, but this is actually all that we need for our current tests on the `AutoComplete` component. The `Editor` component should just accept a content input, which is the main interaction between the two components. In the template of our `MockEditor` component, we just render the content input property. This way, we can easily verify the result of using the `AutoComplete` component.

Let's use this mock editor to write our next test:

```
it('should initialize editor with selected item',
async(inject([TestComponentBuilder], (tcb) => {
    tcb
      .overrideDirective(AutoComplete, Editor, MockEditor)
      .createAsync(AutoComplete).then((fixture) => {
        // Given
        fixture.componentInstance.items = ['one', 'two', 'three'];
        fixture.componentInstance.selectedItem = 'two';
        // When
        fixture.detectChanges();
        // Then
        expect(fixture.nativeElement.textContent.trim())
          .toBe('two');
      });
  })));
```

In our tests, we'd like to test whether the `AutoComplete` component initializes `Editor` (respectively, our `MockEditor` component) with the right content. We test whether `selectedItem` of our `AutoComplete` component successfully reflects into the editor.

We use `TestComponentBuilder`, which creates components asynchronously. Using the `async`

helper function, we tell Jasmine to wait for all asynchronous operations to complete for this test.

Using the `ComponentFixture` that is provided by `TestComponentBuilder`, we can start to interact with the created component. Using the `componentInstance` member of the component fixture, we can set the required input properties of our `AutoComplete` component.

As we're responsible for the triggering of change detection manually in our tests, we use the `detectChanges` method on our fixture to update the component view, based on the new state. This will initiate the change detection lifecycle on our component and perform the necessary view updates.

After the view updates both of our `AutoComplete` component and the underlying `MockEditor` component, we can run our assertions to validate the updated DOM by getting the text content of the `nativeElement` property on our fixture.

For this particular test, we're fine with this approach. However, in other scenarios where we have more DOM elements involved, it wouldn't be sufficient to assert on the root component's `textContent` property directly. This would probably include a lot of noise, which we're not interested in for our assertion. We should always try to narrow our assertion to the fewest details possible.

As we have access to the native DOM element on our fixture, we can simply use the DOM API to select child elements in order to narrow our assertion:

```
expect(fixture.nativeElement.querySelector('ngc-
editor').textContent.trim()).toBe('two');
```

This would successfully select the DOM element of our mock editor, and we can only check the text content inside the editor.

Although this would be a feasible approach, Angular provides us with a much better approach to achieve this goal.

Provided by `ComponentFixture`, we have access to the `DebugElement` tree that is created by `DebugDomRootRenderer` in the context of our test. `DebugElement` allows us advanced inspection of the element tree that was created by Angular when rendering our components. It also contains an advanced querying API, which allows us to search for certain elements in the tree.

Let's rewrite our test to use the advanced capabilities provided by `DebugElement`:

```
...
import {By} from '@angular/platform-browser';
...
  it('should initialize editor with selected item',
async(inject([TestComponentBuilder], (tcb) => {
    tcb
      .overrideDirective(AutoComplete, Editor, MockEditor)
```

```
      .createAsync(AutoComplete).then((fixture) => {
        ...

expect(fixture.debugElement.query(By.directive(MockEditor)).nativeElement.text
Content.trim()).toBe('two');
      });
  }));
```

The `query` and `queryAll` methods that are available on every `DebugElement` object allow us to query the Angular view tree like we would query a DOM tree using `querySelector` and `querySelectorAll`. The difference here is that we can use a predicate helper to query for matching elements. Using the `By` helper class, we can create these predicates, which will then be used in order to query the `DebugElement` tree.

There are currently three different predicates available using the `By` helper:

Member	**Description**
`By.all()`	This is the predicate, which will result in querying for all the child `DebugElement` object of the current `DebugElement` object
`By.css(selector)`	This is the predicate, which will result in querying for `DebugElement` using the specified CSS selector
`By.directive(type)`	This is the predicate, which will result in querying for `DebugElement` that contain the specified directive

Going back to our test, we can now use the query method on the fixture debug element in order to query for our editor. As we've exchanged the real `Editor` component with our `MockEditor` component, we need to query for the latter. We use a `By.directive(MockEditor)` predicate, which will successfully query for the `DebugElement` object that represents the host element of our `MockEditor` component.

The `query` method of the `DebugElement` object will always return a new `DebugElement` object of the first found element if there was a match. It will return `null` if the queried element was not found.

The `queryAll` method of the `DebugElement` will return an array of many `DebugElement` which contains all elements that match the predicate. If there were no matching elements, this method will return an empty array.

Testing component interaction

Although UI interaction testing is probably part of end-to-end testing, we'll look at how to test basic user interaction on your components in this topic.

In this topic, we'll test the autocomplete component behavior if the user clicks on an item in the callout window that shows all available items.

Let's add this test to the already existing `auto-complete.spec.js` module:

```
it('should emit selectedItemChange on click in callout',
async(inject([TestComponentBuilder], (tcb) => {
    tcb
      .overrideDirective(AutoComplete, Editor, MockEditor)
      .createAsync(AutoComplete).then((fixture) => {
        spyOn(fixture.componentInstance.selectedItemChange, 'next');
        fixture.componentInstance.items = ['one', 'two', 'three'];
        fixture.componentInstance.selectedItem = 'one';
        fixture.componentInstance.onEditModeChange(true);
        fixture.componentInstance.onEditableInput('');
        fixture.detectChanges();
        fixture.debugElement
          .queryAll(By.css('.auto-complete__item'))
          .find((item) => item.nativeElement.textContent.trim() === 'two')
          .triggerEventHandler('click');

expect(fixture.componentInstance.selectedItemChange.next).toHaveBeenCalledWith
('two');
    });
  })));
```

First, we want to set up a Jasmine spy on the `selectedItemChange` `EventEmitter` `next` function for our test. This way, we can check later whether our `AutoComplete` component successfully emitted the event when the user selects an item from the callout.

In the `Given` section of our test code, we also call the `onEditModeChanged` and `onEditableInput` methods on the `AutoComplete` component instance. With these calls, we simulate the editor that was used, and there's currently no content in the editor. This will result in the desired filtering, which will present all available items in the callout for selection.

In the `When` section of our code, we first need to trigger change detection on the fixture. This results in the callout with all available auto-complete items being rendered in the `AutoComplete` component.

Now, we can simulate the `click` event on one of our autocomplete items to fishing the actions in this test.

First, we'll select all `DebugElement` object that match the CSS class of our autocomplete items in the callout. This will provide us with an array containing all the elements, where we can

now use the `Array.prototype.find` method to select one specific item based on the contained text.

On the `DebugElement` resulting from our query, we now call the `triggerEventHandler` method to simulate a click event. This will actually not trigger a real click event, but rather it will execute the handler attached to the binding in the view directly.

After simulating a click on the autocomplete item with the text content of `two`, we can now inspect our spy on the `selectedItemChange.next` function. According to the behavior in our component, this should have been called with the selected item value.

Testing user interaction on components is made very easy using the `DebugElement`. We also decouple our tests from the underlying DOM events by taking the shortcut enabled by the `triggerEventHandler` method.

Tip

The `triggerEventHandler` method operates on the virtual element tree of Angular, rather than the actual DOM tree. Due to this, we can also use this method to trigger event handlers that are attached to component output properties.

Testing our plugin system

In the previous sections, we created tests for the `AutoComplete` component, which is a rather simple UI component. However, we learned about all the techniques that are required to perform testing on more complex components or even systems of components.

Now, we'll look into testing the plugin system that was created in Chapter 10, *Making Things Pluggable*.

It's probably a good time to recap on the plugin system architecture overview before working on this topic. As always with testing, it's crucial to understand exactly what's happening in the system under test.

Let's create a new `plugin.spec.js` file in the `lib/plugin` path.

Before we implement our first test function for this subject, we will need to create some dummy components and plugins to test our system with. Let's create these at the top of our testing module:

```
@Component({
  selector: 'dummy-plugin-component-1',
  template: 'dummy1'
})
export class DummyPluginComponent1 {}

@Component({
  selector: 'dummy-plugin-component-2',
  template: 'dummy2'
})
export class DummyPluginComponent2 {}

@Component({
  selector: 'dummy-application',
  template: 'dummy-slot:<ngc-plugin-slot name="dummy-slot"></ngc-plugin-slot>',
  directives: [PluginSlot]
})
export class DummyApplication {}
```

Nothing special here. We declare two dummy components with a static template that will serve us in performing our plugin tests. Additionally, we created a dummy application component, which will be our main testing component. In the following tests, we will make use of a dummy component to test our `PluginSlot` directive, as opposed to testing a component directly.

Next, we'll need to mock our `PluginService` injectable, which is designed to load plugins asynchronously from URLs. In our mock, we'd want to override this functionality. Instead of loading plugins from URLs, we want to load some predefined test plugins:

```
@Injectable()
```

```
export class MockPluginService extends PluginService {
  constructor() {
    super();
    this.change = {
      subscribe() {}
    };
  }

  loadPlugins() {}
}
```

We override the `loadPlugins` method to avoid any plugins being loaded during the construction of the service. We also override the RxJS subject present on the `change` property in order to prevent any reactive behavior of our plugin system because this would only disturb our tests.

Let's dive right into our first test where we want to test a very basic plugin with one plugin component, to be instantiated correctly by the `PluginSlot` directive. First, we set up our test structure using the `describe` and `it` functions:

```
describe('PluginSlot', () => {
  beforeEachProviders(() => [
    provide(PluginService, {
      useClass: MockPluginService
    })
  ]);

  it('should create dummy component into designated
slot',async(inject([TestComponentBuilder, PluginService], (tcb, pluginService)
=> {
      tcb.createAsync(DummyApplication).then((fixture) => {
          ...
        });
}));
```

The only difference here to what we already knew is that we use a new `beforeEachProviders` function from the `@angular/core/testing` module. This function allows us to set up some default providers that are used in our tests. As all our plugin system tests will rely on the presence of `PluginService`, we use this function to set up the mock provider resolving to our `MockPluginService` class.

Instead of using `beforeEachProviders`, we could also use the `overrideProviders` method in the `TestComponentBuilder` to provide additional injectables. However, this will limit the use to the inside of our components. If we want to interact with the service from our test function, we need to use the `beforeEachProviders` helper.

Using the `inject` helper, we inject `TestComponentBuilder` and `PluginService`, which we provided using the `beforeEachProviders` helper.

Let's now implement the missing test body inside of the `Promise` callback after executing

`createAsync`.

As a first step, we define a new dummy plugin, which uses the `PluginConfig` decorator from the previous chapter. We create a `PluginPlacement` in the plugin metadata, which includes a mapping of `DummyPluginComponent1` into the slot with the name `dummy-slot`. If you take a look at the `DummyApplication` component that we use in this test again, you can see that it contains a `PluginSlot` directive with the name attribute set to `dummy-slot`:

```
@PluginConfig({
  name: 'dummy-plugin',
  description: 'Dummy Plugin',
  placements: [
    new PluginPlacement({slot: 'dummy-slot', priority: 1, component:
DummyPluginComponent1})
  ]
})
class DummyPlugin {}
```

This plugin should now cause the `DummyPluginComponent1` component to be rendered in the plugin slot of our `DummyApplication` class.

As a next step, we add the `DummyPlugin` class to the plugins list of our `MockPluginService` mock service:

```
pluginService.plugins = [{
  type: DummyPlugin,
  config: DummyPlugin._pluginConfig,
  instance: new DummyPlugin()
}];
```

The object that we're adding to the plugins array of the `MockPluginService` simply simulates a plugin that would normally be loaded in `PluginService`.

Next, we put aside a reference to the `PluginSlot` directive, which is placed in our `DummyApplication` component. For this, we can use the `query` method on the `DebugElement` root of our fixture. We use a predicate, which allows us to query by the directive type of our `PluginSlot` component:

```
const pluginSlot = fixture.debugElement
  .query(By.directive(PluginSlot))
  .injector
  .get(PluginSlot);
```

We need the reference to the directive instance of the plugin slot in order to initialize the slot prior to our test assertion. This is an important step because we can't rely on the observable subject in our `MockPluginService` class to initialize our `PluginSlot` directive. We explicitly disabled the reactive features of our plugin system in order to perform proper testing. Therefore, we need to manually initialize our plugin slot before we can perform any assertion.

After executing the query with the directive predicate (searching for an element which contains the `PluginSlot` directive), we'll receive `DebugElement` of our plugin slot element. In order to get the directive instance, we use the element injector present on each `DebugElement` object.

The `initialize` method on the `PluginSlot` component instance will create all relevant plugin components. Luckily, this will also return a `Promise` to us, which will be resolved once all components have been created in the view of our `ApplicationDummy` component:

```
pluginSlot.initialize().then(() => {
  fixture.detectChanges();
  expect(fixture.nativeElement.textContent).toBe('dummy-slot:dummy1');
});
```

In the callback of the `Promise` returned by the `initialize` method of the `PluginSlot` instance, we can finally assert on the text content of the root element of our `DummyApplication` component.

As the `DummyPluginComponent1` class has a simple static template that contains the text `dummy1`, we should see a complete text content of `dummy-slot:dummy1` in our application view.

This is it for our first plugin test. Now, we will look at a second test, which we'll use to verify another feature of our plugin system. Our plugin system should also be able to render two components of the same plugin into two separate plugin slots. However, in the template of our `DummyApplication` component, we currently only have one plugin slot with the name `dummy-slot`.

In order to modify the template of our `DummyApplication` component just for a particular test, we can use the `overrideTemplate` method on `TestBuilderComponent`:

```
  it('should create two dummy components of same plugin into different
slots',async(inject([TestComponentBuilder, PluginService], (tcb, pluginService)
=> {
      const template = 'dummy-slot1:<ngc-plugin-slot name="dummy-slot1">
</ngc-plugin-slot>dummy-slot2:<ngc-plugin-slot name="dummy-slot2"></ngc-
plugin-slot>';

      tcb.overrideTemplate(DummyApplication, template)
        .createAsync(DummyApplication).then((fixture) => {
          ...
        });
    }))
  );
```

In our test function, we create a new template for our `DummyApplication` component. We're adding two plugin slots to the template with their name attributes set to `dummy-slot1` and `dummy-slot2`.

Now, we can use the `overrideTemplate` method on `TestComponentBuilder` to override the

`DummyApplication` component template before we create it. This provides us with the necessary flexibility to reuse mock and dummy components for different tests.

Let's take a look at the code that comes in the `createAsync` promise callback:

```
@PluginConfig({
  name: 'dummy-plugin',
  description: 'Dummy Plugin',
  placements: [
    new PluginPlacement({slot: 'dummy-slot', priority: 1, component:
DummyPluginComponent1}),
    new PluginPlacement({slot: 'dummy-slot', priority: 2, component:
DummyPluginComponent2})
  ]
})
class DummyPlugin {}
```

First, we create a new `DummyPlugin` plugin class and use the `PluginConfig` decorator to configure it. In the placement metadata, we configure the mappings so that we map two components into different plugin slots. The first component is mapped to the plugin slot with the name `DummySlot1`, while the second one will go to the slot with the name `DummySlot2`. We've overridden our `DummyApplication` template to include both of these plugin slots.

We now add our `DummyPlugin` class to the `MockPluginService` class and simulate the plugin being loaded:

```
pluginService.plugins = [{
  type: DummyPlugin,
  config: DummyPlugin._pluginConfig,
  instance: new DummyPlugin()
}];
```

The following code queries for `DebugElements` in the fixture using the `queryAll` method. We use a predicate that queries for all elements containing the `PluginSlot` directive. With an additional call to `Array.prototype.map`, we transform the array in order to get back the component instances of the discovered `PluginSlot` components directly:

```
const pluginSlots = fixture.debugElement
  .queryAll(By.directive(PluginSlot))
  .map((debugElement) => debugElement.injector.get(PluginSlot));
```

Now, it's time to complete our test. Using the `Promise.all` function, we're able to streamline an array of Promises into a single Promise, which will resolve once all underlying Promises are resolved. We can then map our `pluginSlots` array by executing the initialize method on each of the `PluginSlot` components. This will return an array of Promises to us, which will resolve when all components within the plugin slots are created:

```
Promise.all(
  pluginSlots.map((pluginSlot) => pluginSlot.initialize())
).then(() => {
  fixture.detectChanges();
```

```
  expect(fixture.nativeElement.textContent).toBe('dummy-slot1:dummy1dummy-
slot2:dumm
y2');
});
```

In the `then` callback of the consolidated promise using the `Promise.all` function, we can finally perform our assertion. With the overridden template of our `DummyApplication` component and the output of our two plugin components in the two separated plugin slots, we should get a text content of `dummy-slot1:dummy1dummy-slot2:dummy2`.

This is the last test that we look at in this chapter. However, there are more tests in the code that comes with this book. Just check out the code repository and get your hands on those tests yourself.

Summary

In this chapter, we learned how to write concise unit tests for our components. We followed a BDD style approach of writing tests, and we also covered the basics of the JavaScript testing framework, Jasmine.

We learned about the debugging tools that are available in Angular and how to set up an injector environment for testing. Using `TestComponentBuilder`, we were able to perform tests in a very flexible but precise way. We also learned about the view tree of multiple `DebugElement` that are created along with `TestComponentBuilder` running in the debug environment. This allowed us to perform clever inspection and apply practical queries to the rendered views in order to assert expected results.

We used the `inject` and `async` helpers to inject dependencies and, at the same time, run asynchronous tests. We built both mock and dummy components in order to isolate our tests from the rest of our application.

Appendix A. Task Management Application Source Code

The source code of the task management application built in this book is available from the Packt download servers. The links for each chapter's final code are listed in this appendix. This appendix also provides instructions on how to use the downloaded code and the steps required in order to run the code. Furthermore, it helps you troubleshoot some of the common problems that you can experience when working with the task management application and some hints on how to resolve them.

Download

The following list provides download links for each chapter of the book. The download links reference downloadable archive files, which need to be unpacked on your local hard drive:

Chapter	**Link**
Chapter 2, *Ready, Set, Go!*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-2
Chapter 3, *Composing with Components*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-3
Chapter 4, *No Comments, Please!*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-4
Chapter 5, *Component-Based Routing*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-5
Chapter 6, *Keeping Up with Activities*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-6
Chapter 7, *Components for User Experience*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-7
Chapter 8, *Time Will Tell*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-8
Chapter 9, *Spaceship Dashboard*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-9
Chapter 10, *Making Things Pluggable*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-10
Chapter 11, *Putting Things to the Test*	https://github.com/PacktPublishing/Mastering-Angular-2-Components/tree/master/angular-2-components-chapter-11

The complete source code is also available at http://www.packtpub.com.

Prerequisites

The task management application is built using Node.js technologies; therefore, it requires that you have Node.js installed on your machine before you can run any of the code.

You can download and install Node.js from the website, http://nodejs.org. In order to build the Sass source files and start a server with live-reload, two global node modules are required:

```
npm install -g gulp live-server
```

Usage

After downloading the source code of an individual chapter, you'll need to install NPM as well as JSPM dependencies. By running the following two lines of code on your console, you'll make sure all dependencies are installed. Make sure that you run these commands from inside the downloaded and extracted code folder:

```
npm install
jspm install
```

After you've installed the required dependencies, you can go ahead and start the application:

```
npm start
```

The NPM start script will invoke gulp in order to compile any Sass files as well as live server to start a static server with live-reload. Please read the prerequisites topic on how to install these global Node.js modules.

Troubleshooting

We've carefully considered various environments where the task management source code will be executed. However, there's always a chance that you'll experience some issues when working with the source code. This topic will provide you with solutions for common problems when working with the task management application.

Cleaning IndexDB to reset data

The task management application uses a data store that is persisted in your browser. If you use the application extensively across multiple chapters, chances are high that your data is causing some application instability.

If you'd like to clean the local database used in the application, you can run the following line of code on the debugger console of your browser. It's important that you run the code snippet in the debugger of your browser with the task management application open. If your browser points to a different origin, the application database can't be deleted. After deleting the database, the application can be reloaded, and it will recreate the database with the initial sample data:

```
indexedDB.deleteDatabase('_pouch_angular-2-components');
```

Enabling web components in Firefox

Chapter 6, *Keeping Up with Activities*, relies on the presence of Shadow DOM in the browser. Chrome supports Shadow DOM natively after version 35. In Firefox, you can enable Shadow DOM by visiting the `about:config` page and turning on the `dom.webcomponents.enabled` flag.

IE, Edge, and Safari don't support this standard at all; however, we can teach them how to deal with Shadow DOM by including a polyfill named `webcomponents.js`. You can find more information on this polyfill at https://github.com/webcomponents/webcomponentsjs.

Index

A

B

C

- about / Custom UI elements, Recap
- checkbox / Custom UI elements
- toggle buttons / Custom UI elements

D

E

G

- Gang of Four (GoF)
  - about / Decorators

H

- hello world component
  - writing / Your first component
  - writing, JavaScript used / JavaScript of the future
- HTML
  - URL / Building SVG components

I

J

L

M

N

O

P

R

S

T

U

V

W