CREATING **IONIC** APPLICATIONS WITH **ANGULAR**

JOSHUA MORONY

Changelog

Version 24 - RELAUNCH

- Renamed the book to **Creating Ionic Applications with Angular**
- Rewrote large sections of the book
- Minor refresh/updates for sections that were not rewritten
- All 5 example applications have been rewritten/refactored
- `async/await` is now used throughout the book instead of the standard promise syntax
- Updated the iOS building/submission steps to use App Store Connect
- Updated the Android building/submission steps to use App Bundles instead of an APK
- Refilmed the bonus video course available in the complete package
- Refreshed/refactored the Fliqs and Refine themes

Version 23 - updated for Ionic 5

Version 22 - updated for Angular 8

- Minor typographical fixes
- Minor code changes
- Modified backend for chat application in Expert package
- New cover!

Version 21 - updated for the Ionic 4 Stable Release

- Added 'Ion' prefix to Ionic component references, e.g. IonContent and IonList instead of Content and List
- Changed navigation links from href to routerLink
- Modified tabs for breaking changes
- Minor typographical fixes
- Minor code changes

Version 20 - updated for Ionic 4, Beta 7

- Updated goRoot, goBack, and goForward methods to navigateRoot, navigateBack, and navigateForward
- Updated loading overlay to use 'message' instead of 'content'
- modal.onDidDismiss() updated to handle promise
- Changed method for scrolling content area to bottom
- Updated code syntax highlighting for better contrast/readability
- Minor typographic fixes

Version 19 - updated for Ionic 4 (Beta)

- Updated book for Ionic 4
- Updated applications for Ionic 4
- Updated themes for Ionic 4 (Premium + Expert only)
- Updated video course for Ionic 4 (Expert only)
- Various improvements for all example applications
- Simplified Code Signing Steps for App Store Distribution
- Capacitor is now used in place of Cordova

- Ionic 3 Legacy version still available on request

Version 18 - updated for Ionic 3.2.0

- Updated commands for Ionic CLI v3
- Minor bug fixes and updates

Version 17 - updated for Ionic 3.0.0

- Updated basics section to include information on lazy loading and the new @IonicPage
- Updated example application to use lazy loading
- Minor bug fixes and updates

Version 16 - updated for Ionic Native 3.x

- Updated Ionic Native integrations to Ionic Native 3.x
- Minor updates

Version 15 and older notes removed for brevity

Table of Contents

Introduction

1. Welcome
2. An Overview of the Ionic Ecosystem
3. Basic Concepts
4. Ionic & Angular Syntax
5. Native Builds and Functionality with Capacitor

Ionic Basics

6. Lesson 1: Generating an Ionic Application
7. Lesson 2: Anatomy of an Ionic Project
8. Lesson 3: Ionic CLI Commands
9. Lesson 4: Decorators
10. Lesson 5: Classes
11. Lesson 6: Templates
12. Lesson 7: Styling and Theming
13. Lesson 8: Navigation, Routing, and Lazy Loading
14. Lesson 9: User Input
15. Lesson 10: Saving Data
16. Lesson 11: Fetching Data, Observables, and Promises
17. Lesson 12: Native Functionality

Quick Lists

18. Lesson 1: Quick Lists Introduction
19. Lesson 2: Quick Lists Getting Ready
20. Lesson 3: Basic Layout
21. Lesson 4: Creating an Interface and Checklist Service
22. Lesson 5: Creating Checklists and Checklist Items
23. Lesson 6: Saving and Loading Data
24. Lesson 7: Creating an Introduction Slider & Theming
25. Quick Lists Conclusion

Giflist

26. Lesson 1: Giflist Introduction
27. Lesson 2: Giflist Getting Ready
28. Lesson 3: The List Page
29. Lesson 4: The Reddit API and HTML5 Video
30. Lesson 5: Integrating the Data Service
31. Lesson 6: Settings
32. Lesson 7: Styles
33. Giflist Conclusion

Snapaday

34. Lesson 1: Snapaday Introduction
35. Lesson 2: Snapaday Getting Ready
36. Lesson 3: The Layout
37. Lesson 4: Taking Photos with the Camera
38. Lesson 5: Saving and Loading Photos
39. Lesson 6: Custom Pipe and Flipbook

40. Lesson 7: Integrating Local Notifications
41. Lesson 8: Style
42. Snapaday Conclusion

Camper Mate

43. Lesson 1: Camper Mate Introduction
44. Lesson 2: Camper Mate Getting Ready
45. Lesson 3: Creating a Tabs Layout
46. Lesson 4: User Input and Forms
47. Lesson 5: Implementing Google Maps and Geolocation
48. Lesson 6: Saving and Retrieving Data
49. Lesson 7: Styling
50. Camper Mate Conclusion

Camper Chat

51. Lesson 1: Camper Chat Introduction
52. Lesson 2: Camper Chat Getting Ready
53. Lesson 3: Login Page and Sliding Menu Layout
54. Lesson 4: Facebook Authentication
55. Lesson 5: Creating Messages & Navigation
56. Lesson 6: Using Firestore to Store Messages
57. Lesson 7: Firestore Security Rules
58. Lesson 8: Styling & Animations
59. Camper Chat Conclusion

Testing and Debugging

60. Testing & Debugging

Building and Submitting

61. Preparing Assets
62. Building for iOS and Distributing to the Apple App Store
63. Building for Android and Distributing to Google Play
64. Building for iOS and Android Using Ionic Package
65. Publishing as a PWA (Progressive Web Application) with Firebase

Conclusion

66. Conclusion

Introduction

Welcome!

Hello and welcome to **Creating Ionic Applications with Angular!** This book will teach you everything you need to know about Ionic, from the basics right through to building an application for iOS and Android and submitting it to app stores.

People will have varying degrees of experience with Ionic and/or Angular when reading this book. Whatever your skill level is, it should not matter too much. All of the lessons in this book are thoroughly explained and make no assumption of previous experience with Ionic or Angular.

However, this book does not contain an introduction to HTML, CSS, and JavaScript. You should have a reasonable amount of experience with these technologies before starting this book. If you need to brush up on your skills with these technologies I'd recommend taking a look at the following:

- Learn Javascript

This book has many different sections, but there are three distinct areas. We start off with the **basics**, we then progress onto some **application walkthroughs** and then we cover **building and submitting** applications.

All of the example applications included in this course are completely standalone. Although, in general, the applications increase in complexity a little bit as you go along, I make no assumption that you have read the previous walkthroughs and will explain everything thoroughly in each example.

NOTE: If you have purchased a package which includes the video course, I would recommend watching it *before* reading the book. It is not required, but it is a basic introductory level course so it makes for more of a logical progression to watch it first.

Avoid Copying & Pasting

Ultimately, it is up to you to decide how you learn best. However, one piece of advice I have enjoyed following is to always manually type out examples when learning. When I just copy and paste examples instead of typing them out myself, I find that when the time comes to write my own code I always end up having to go back to an example, again and again, to see what to do.

By typing out the examples yourself you will start getting that muscle memory trained up, and by more actively engaging with the exercise you will likely be able to commit the concepts, syntax, and keywords more effectively to memory. I see it as kind of like writing out a cheat sheet for an exam (for those of you who have sat exams where you are allowed to bring in a sheet of paper with whatever you like on it): often if you write out the cheat sheet manually, you barely even need to look at it in the exam because it's all stored in your head.

It is much slower and much more tedious, but you will likely get more out of this book if you do it that way. You will likely mistype some things, use the wrong syntax, and run into errors but this is a good thing. I also find that there is no better way to enforce some

concept in your head than by running into an error and then figuring out where things went wrong.

Text Editor/IDE

I generally use Visual Studio Code for development, but you can use whatever text editor or IDE you like to progress through this book. The instructions in this book will be kept generic such that it does not matter whether you are using Sublime Text, Visual Studio Code, or anything else. However, if you don't already have a strong preference, I would definitely recommend giving Visual Studio Code a try.

Disclaimer

Please keep in mind that this book is purely for educational purposes. You are welcome to use examples from this book for your own real-world applications if you wish, but no guarantees are given for the code that is supplied. You should always do your own research as to whether solutions are suitable for your particular circumstances.

Updates & Errata

Ionic is constantly being developed, and so some changes to what you see in this book are inevitable. Most of what you read in this book won't change, but there are still minor changes from time to time. I will be updating this book for each new major Ionic release, with updates between major releases if required, and **you will receive these updates for free**. Any time I update the book you should receive an email notification with a new download link.

I'll be keeping a close eye on changes and making sure everything works, but it's a big

book so if you think you have found an error **please email me** and I'll update it as soon as I can.

Breaking Changes

As I mentioned, minor/patch updates between major version releases usually won't contain breaking changes and so you won't need to change your code. However, it is still possible for a breaking change to be introduced at some point between major versions, so it's always a good idea to keep an eye on this list which details bug fixes/features/changes for each release. Take note if you see any releases that contain **BREAKING CHANGES**.

Screenshots

Many screenshots are used throughout this book to give you a quick look at the general appearance of the applications we are building or of various tools/services we might be using. Please keep in mind that there may be some differences between what you see in the screenshot and what you see on your own machine - small things will often change and I generally won't update the images in this book each time they do.

Conventions Used in This Book

The layout used in this book doesn't require much explaining, however, you should look out for:

> Blocks of text that look like this

As they are actions you have to perform. For example, these blocks of text might tell you to create a file or make some code change. You will mostly find these in the application

walkthroughs. This syntax is useful because it helps distinguish between code changes I want you to make to your application and just blocks of code that I am showing for demonstration purposes.

NOTE: You will also come across blocks of text like this. These will contain little bits of information that are related to what you are currently doing.

IMPORTANT: You will also see a few of these. These are important "Gotchas" you should pay careful attention to.

Ok, enough chat! Let's get started. Good luck and have fun!

An Overview of the Ionic Ecosystem

There was a time when life was simple and we didn't need to worry about making any decisions. Ionic was built on top of Angular, and we used both of these frameworks to build mobile applications. Since the release of Ionic 4, Ionic has been based on web components that work just about *everywhere*. We will talk a little more about what a web component is later, but basically, it is a way to create your own custom HTML elements that work anywhere on the web. The key difference here is that since Ionic is now built on top generic web technologies, it doesn't *require* the Angular framework anymore.

We can still use Ionic/Angular just as we could before, but because of web components now we can also use those same Ionic components in other frameworks (like Vue or React), or with no framework at all.

This is fantastic for people who want to develop with Ionic, but now that there are so many options it does lead to some confusion initially. There are now so many "players" in the Ionic ecosystem now. Some terms you might come across in your learning journey are:

- Ionic
- StencilJS
- Web components
- Angular
- @ionic/angular

- @ionic/core
- React
- Vue
- ..and so on

It can be hard to figure out what you should be using, which frameworks or tools are necessary for the approach you take, and which ones are not. To help explain how everything fits together, I made this little chart:

Although there are a lot of different parts that make up the Ionic ecosystem, the only bits we really need to worry about or learn anything about are the sections highlighted in green above. Even though we may not need to use parts of the ecosystem, it does help to have a broad understanding of what is going on.

StencilJS is a tool that creates web components, and the Ionic team have created a

collection of web components with Stencil which have been packaged as **Ionic Core**. Ionic Core contains the set of web components that make up Ionic's user interface library (lists, buttons, cards, and so on). You will use this core set of components no matter where you are using Ionic (Angular, React, Vue, no framework, etc.), but you don't need to use or understand Stencil itself in order to use the components created by Stencil. The resulting web components that StencilJS creates are just generic web components.

As the graph shows, we have the web components created by Stencil going into this "Ionic Core" package, but Stencil can also be used to create web components for any purpose. You might want to use Stencil to create your own web components to use in your Ionic applications, or you might just want to create web components for some other purpose. Although Stencil is a very cool tool for building web components (and entire applications), **you don't need to know how to use Stencil in order to use Ionic**.

The bundle of web components that is known as "Ionic Core" can then be used by any framework, and it can also be used without a framework at all. There is also a special library called `@ionic/angular` which is the Ionic package that is specific to Angular (the one that Ionic developers have been using for a long time). I will often use "Ionic/Angular" to refer to this package, or also just to Ionic applications built using Angular. This package provides extra functionality that can only be used in Angular, but it will still use the same Ionic web components under the hood. You can consider the general purpose of the `@ionic/angular` package to be to make using Ionic with Angular much smoother and nicer.

To clarify all of this, if we are building an Ionic application without a framework we would use `@ionic/core` directly. If we are using Angular to build Ionic applications we would use `@ionic/angular` which makes use of `@ionic/core` behind the scenes. If we are

using React, we would use `@ionic/react` which also uses `@ionic/core` behind the scenes (and so on).

This explanation may still leave you a little confused, especially if you aren't already familiar with these tools and frameworks. It is not really important to understand all of this. We will be building Ionic/Angular applications, so there are only three parts of that chart that we need to worry about:

- Ionic Core (`@ionic/core`)
- Ionic/Angular (`@ionic/angular`)
- Angular (`@angular/...`)

If you like, you can just pretend the rest of it doesn't even exist. Since Angular has always been the "default" framework for Ionic, there is a lot of support for this particular approach and that is how most Ionic applications are currently made, which is why we are focusing on that approach in this book.

You might wonder why we would bother to use a framework like Angular when we can use Ionic without a framework. Ionic provides the user interface components to make building a mobile application easier, but we often still want to use a framework to help build the event/data/logic side of the application itself. Frameworks give us a nice and organised structure to work within, as well as a bunch of built-in features to make development easier. A lot of the time, developers have a preference for a particular framework for various reasons, and Ionic can generally be incorporated no matter what that preference is.

You don't really *need* to know about the other options out there, but I do want to briefly cover some of the different options available and when you might use them. This should

also help clarify why we are specifically using Ionic with the Angular framework.

Using Ionic with Angular

Historically, Ionic was a UI (User Interface) framework built on top of Angular. Ionic provided a bunch of Angular components that could be used to easily create native style mobile interfaces, which meant that in order to use Ionic you also needed to use Angular. Now that the Ionic components are built using generic web components (which can be used anywhere), Angular is no longer *required* to build applications with Ionic. That doesn't mean that we shouldn't use Angular, though. There is a lot more to an application than just the user interface elements that we see on the screen, there is a lot of stuff that goes on behind the scenes, and Angular provides a lot of things out of the box to help us build a well-structured application.

Some things the Angular framework provides for us are:

- An organised project structure
- Change Detection - so that the views in your application update when data changes
- Two-way Data Binding - so that changing a value in your views will automatically update the underlying data (and vice versa)
- Providers - an object that can be used throughout your application to provide helper methods, or share data
- Helper Libraries - like the HttpClient for making HTTP requests

These types of features are something that most frameworks provide to one degree or another (often frameworks will take slightly different approaches, though). However, there are a few reasons that I think that Angular makes a good default framework for Ionic applications.

- Since Ionic applications have historically been built with Angular, there is a lot of community content available (like the 100s of Ionic/Angular tutorials on my blog)
- Most of the functionality you need is provided out of the box, so you don't need to worry about finding other libraries to handle various aspects of your application, or coding your own solutions
- Ionic has built-in support for Angular through the `@ionic/angular` package, which makes Ionic/Angular integration a little easier
- Angular is heavily optimised for mobile
- There are a lot of 3rd party libraries available that can easily be combined with Angular if necessary

At this time, I think that Ionic/Angular is a good default option for most people given the amount of community support available. However, if you already have knowledge in a different area (e.g. React) it might be worth considering that as well.

Using Ionic with a Different Framework

Although I think that there are a lot of benefits to using Angular, you can do whatever you want! Integrating Ionic with any other framework is as simple as installing the "Ionic Core" web components, and making sure that you enable support for web components/custom elements in that framework. For some frameworks like React and Vue, you will also find that additional packages are available to help integrate Ionic into these frameworks.

Using Ionic with Stencil

This is perhaps the most "pure" approach to building Ionic applications, as it is just straight up Ionic and web tech - no external frameworks or libraries required (Ionic itself isn't really a framework so to speak, it is just a collection of web components).

Stencil is the web component compiler that Ionic uses to build the web components that make up the Ionic library (we will talk more about web components later). Generally, we don't need to worry about Stencil as it is just something that is used behind the scenes by Ionic. However, if you want to, you could build your entire application using Stencil - you use the web components Ionic provides, and then you build your own web components with Stencil to make up the rest of the application.

Although you miss out on some of the built-in features that make development easier, the main benefit to this approach is that it doesn't require any framework libraries (in fact, you could build out your entire application with no dependencies at all).

I would recommend this approach to people who are willing to put in a bit more development effort to really push the limits of performance, and especially if you are targeting users who may have low-end devices or you need to cater for slow network speeds or expensive data. Also if you just like the idea of not using a specific framework. If you are interested in learning to use Ionic with StencilJS, I have another book on that topic: Creating Ionic Applications with StencilJS.

Basic Concepts

Depending on your level of experience, you may or may not already understand some of the concepts that I am about to mention. Understanding these concepts is important to being able to understand how to use Ionic, so I will be explaining these concepts as if you just have a basic understanding of "normal" ES5 JavaScript (although more modern JavaScript is quickly becoming more widely adopted).

Web Components

One of the key benefits of using Ionic is how easy it is to add complex mobile interface elements to the pages in your application. In Ionic, the templates that create the pages/views in your application will look something like this:

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-buttons slot="end">
      <ion-button (click)="doSomething()"><ion-icon slot="icon-
only" name="play"></ion-icon></ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

```
<ion-content>

  <ion-list>

    <ion-item-sliding *ngFor="let photo of photoService.photos">

      <ion-item>
        <img [src]="photo.path" />
        <ion-badge slot="end" color="light">{{photo.dateTaken |
daysAgo}} days ago</ion-badge>
      </ion-item>

      <ion-item-options>
        <ion-item-option color="light"
(click)="photoService.deletePhoto(photo)"><ion-icon slot="icon-
only" name="trash"></ion-icon></ion-item-option>
      </ion-item-options>

    </ion-item-sliding>

  </ion-list>

</ion-content>
```

What are all these strange non-HTML standard tags? These are the **web components** that Ionic provides. The basic idea is that we can build our own custom HTML tags to easily drop in functionality in a web page. This is what the core Ionic framework provides, a bunch of web components that provide the user interface elements required to create a

mobile application like toolbars, lists, buttons, and so on. Instead of spending days coding up a high-performance scrollable list that works well on mobile, we just drop in `<ion-list>`.

There is also some Angular specific syntax included in the example above like `*ngFor` and `(click)` but we will discuss that later.

First, let's talk a little bit more about Web Components before we move on, as they are kind of a big deal. Web Components are not specific to Ionic or Angular, they are becoming a new standard on the web to create modular/self-contained chunks of code that can easily be inserted into a web page (kind of like Widgets in WordPress).

> "In a nutshell, they allow us to bundle markup and styles into custom HTML elements." - Rob Dodson

Rob Dodson wrote [a great post on Web Components] where he explains how they work and the concepts behind it. He also provides a really great example, and I think it really drives the point home of why Web Components are useful.

Basically, if you wanted to add an image slider as a web component, the HTML for that might look like this:

```
<img-slider>
  <img src="images/sunset.jpg">
  <img src="images/arch.jpg">
  <img src="images/grooves.jpg">
```

```
  <img src="images/rock.jpg">
</img-slider>
```

instead of (without web components) this:

```
<div id="slider">
  <input checked="" type="radio" name="slider" id="slide1"
selected="false">
  <input type="radio" name="slider" id="slide2" selected="false">
  <input type="radio" name="slider" id="slide3" selected="false">
  <input type="radio" name="slider" id="slide4" selected="false">
  <div id="slides">
    <div id="overflow">
      <div class="inner">
        <img src="images/rock.jpg">
        <img src="images/grooves.jpg">
        <img src="images/arch.jpg">
        <img src="images/sunset.jpg">
      </div>
    </div>
  </div>
  <label for="slide1"></label>
  <label for="slide2"></label>
  <label for="slide3"></label>
  <label for="slide4"></label>
</div>
```

Rather than downloading some jQuery plugin and then copying and pasting a bunch of HTML into your document, you could just import the web component and add something simple like the image slider code shown above to get it working. All of the tricky stuff will be handled behind the scenes. Although web components are a general concept that applies to all of the web, this is exactly the concept behind Ionic - Ionic handles all the tricky stuff behind the scenes and we just drop the web components into our application.

Web Components are super interesting, so if you want to learn more about how they work (e.g. The Shadow Dom and Shadow Boundaries) then I highly recommend reading Rob Dodson's post on Web Components. However, a deep knowledge of web components is not required to use Ionic.

ECMAScript 6 (ES6)

Before we talk about ECMAScript 6, we should probably talk about what ECMAScript even is. There's quite a bit of history involved which we won't dive into, but for the most part: **ECMAScript** is a standard, **JavaScript** is an implementation of that standard. ECMAScript defines the standard and browsers implement it. In a similar way, HTML specifications (most recently HTML5) are defined by the organising body and are implemented by the browser vendors. Different browsers implement specifications in different ways, and there are varying amounts of support for different features, which is why some things work differently in different browsers.

The HTML5 specification was a bit of a game changer, and in a similar way so is the ECMAScript 6 specification. It brings about some pretty drastic changes to the way we code with JavaScript and in general, will make Javascript a much more mature language

that is capable of more easily creating large and complex applications (which JavaScript was never really originally intended to do).

We're not going to go too much into ES6 here, because you will learn what you need to know throughout the book, but I will give a few examples to give you a sense of what it actually is. Some features ES6 introduced to Javascript are:

Classes

```
class Shape {
    constructor (id, x, y) {
        this.id = id
        this.move(x, y)
    }
    move (x, y) {
        this.x = x
        this.y = y
    }
}
```

This is a big one, and something you would be familiar with if you have experience with more traditional programming languages like Java and C#. People have been using class-like structures in Javascript for a long time through the use of functions, but there has never been a way to create a "real" class. Now there is. If you don't know what a class is, don't worry, there is an entire lesson dedicated to it later.

In Ionic, classes like this are used to power the logic for the pages in our application. A typical class might look something like this:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor() {

  }

  someFunction() {
    // do something
  }

}
```

These classes will be responsible for doing things like handling events when users click on a button, setting up the data for a page, or making calls to other services to perform some function. The class that we are using as an example here also has some extra Angular specific syntax like the `@Component` decorator, but we will be getting to that later.

Modules

```
//  lib/math.js
// Exports a function called "sum" that can be used elsewhere
export function sum (x, y) { return x + y }
// Exports a variable called "pi" that can be used elsewhere
export var pi = 3.141593 //

//  someApp.js
// Make everything that was exported in lib/math available in this
file
import * as math from "lib/math"
// Use the "sum" method and "pi" variable that was imported
console.log("2PI = " + math.sum(math.pi, math.pi))

//  otherApp.js
// We can reuse those methods/variables in other files as well
import { sum, pi } from "lib/math"
console.log("2PI = " + sum(pi, pi))
```

Modules allow you to modularise your code into packages that can be imported anywhere you need in your application, this is something that is going to be heavily used in Ionic. We will get into this later, but essentially any components (classes) we create in our application we "export" so that we can "import" them elsewhere.

Promises

Promises are important for handling **asynchronous** behaviour. This can be a tricky concept to get initially. Not every action we perform in our code happens right away, or **synchronously**. If we want to add two numbers together or check a condition, these actions can be performed instantly. This means that we can perform the action, and then continue on with executing the rest of the code.

However, sometimes we may want to do things like retrieve data from a server, or perform some other action that could take anywhere from a few hundred milliseconds to 10 or more seconds to complete. If we were to perform these actions **synchronously**, that means we would need to **wait** for them to complete **before** our application could move on. The application would essentially be frozen whilst we wait for the operation to complete, which we definitely do not want.

Instead, we can perform a task **asynchronously** with **promises**. This allows the task to run in the background whilst the application continues operating as normal. Then when the operation completes, we can handle the response like this:

```
doSomething().then((response) => {
  console.log(response);
});
```

We use the `then` statement to handle the response the function returns. In this case, `doSomething` is a function that returns a promise. This means that when this function is triggered, the application won't need to wait until the `response` is available before continuing. When `response` is available, we can do whatever we want with it then.

It is important to understand the "time flow" of what is happening, and people often run into trouble with that. Let's consider the following example:

```
this.name = 'Name not set yet';

getUsersName().then((response) => {
  this.name = response;
});

displayUsersName();
```

In this example, we are trying to retrieve the user's name and then display it. Let's assume that this is an **asynchronous** operation, and that the name needs to be loaded from a server. The code above **will not work**.

This is perhaps the most common mistake I see, and it stems from not understanding how asynchronous code is executed. Here is what actually happens:

1. Set `this.name` to `Name not set yet`
2. Trigger the `getUsersName` promise that will fetch the users name
3. Call the `displayUsersName` function to display the user's name (Name not set yet)

   ... waiting for promise to finish ...
4. Set `this.name` to the response from the promise

As you can see, the promise is triggered, but then the rest of the code executes without waiting for the promise to finish (as it should). The issue is that we are trying to display the

user's name before it is set, and the result will be "Name not set yet". So, how do we handle this properly? If your code relies on the result of a promise, you need to make sure that you trigger that code as a result of the promise handler.

Here is the code that we *should* use:

```
this.name = 'Name not set yet';

getUsersName().then((response) => {
  this.name = response;
  displayUsersName();
});
```

and the flow for this code would look like this:

1. Set `this.name` to `Name not set yet`
2. Trigger the `getUsersName` promise that will fetch the users name ... waiting for promise to finish ...
3. Set `this.name` to the response from the promise
4. Call the `displayUsersName` function to display the user's name

If this concept is new to you, it's probably going to take a little getting used to, as it is kind of hard to wrap your brain around initially. If you run into issues with promises in your applications (like undefined variables that you expect to be defined), and you probably will, try to work out the "time flow" as I've described above. Remember that any `.then` handlers are going to trigger *after* the rest of your synchronous code has run. It is often

also helpful to go into your code and add comments to indicate the order of execution.

Async/Await

The syntax we have just discussed for promises above is the "standard" syntax, but we can also use `async` and `await` to handle promises. I like using the standard promise syntax to explain the general concept, but using `async` and `await` is *much* easier in practice and it is what we will be using in this book.

The basic idea is that rather than creating a handler that is passed the result of the asynchronous operation like this:

```
doSomething().then((response) => {
  console.log(response);
});
```

We can do this instead:

```
const response = await doSomething();
console.log(response);
```

Any time we want to get the value from a resolved promise, we can just `await` it. It is important to know that the `await` keyword can **only be used inside of an async method**.

That means that if we tried to do this:

```
myMethod(): void {
  const response = await doSomething();
  console.log(response);
}
```

...it would not work. However, if we mark `myMethod` as being `async`:

```
async myMethod(): Promise<void> {
  const response = await doSomething();
  console.log(response);
}
```

...it will work. Notice that the return type of `void` (meaning the function does not return anything) has been changed to `Promise<void>` (meaning the method returns a promise that does not resolve with a value) - we will discuss types more in just a moment. Any time we mark a method as `async` it will automatically wrap whatever that method returns in a promise (even if it does not return anything).

To further highlight the usefulness of `async` and `await` let's go back to the example we discussed before:

```
this.name = 'Name not set yet';

getUsersName().then((response) => {
  this.name = response;
  displayUsersName();
});
```

I mentioned that it is important to trigger `displayUsersName()` *inside* of the promise handler, otherwise `displayUsersName()` would be called before `this.name = response` is called, due to the way asynchronous code works (promises get executed later, synchronous code gets executed right away). With `async` and `await` that example would look like this:

```
async myMethod() {
  this.name = 'Name not set yet';

  this.name = await getUsersName();
  displayUsersName();
}
```

This code above will work, as the `await` keyword will cause our code to wait until `getUsersName()` has resolved before it continues on to `displayUsersName()`. Not having to write handler functions already makes the code a lot nicer, but it is even more noticeable when we are waiting for several promises. For example, let's say we have some code that first gets a users `id`, then it gets their `name`, and then it calls an `order` method.

With the standard promise syntax that might look something like this:

```
myMethod(){
  this.getUserId().then((id) => {
    this.getUserName(id).then((name) => {
      this.createOrder(id, name).then((result) => {
        console.log(result);
      })
    })
  })
}
```

With `async/await` we could do the same thing like this:

```
async myMethod(){
  const id = await this.getUserId();
  const name = await this.getUserName(id);
  const result = await this.createOrder(id, name);
  console.log(result);
}
```

Block Scoping

Currently, if you define a variable in Javascript it is available anywhere within the function

that it was defined in. The new block scoping features in ES6 allows you to use the `let` keyword to define a variable only within a single block of code like this:

```
for (let i = 0; i < a.length; i++) {
    let x = a[i];
}
```

If I were to try and access the `x` variable outside of the for loop, it would not be defined. In general, it is a good idea to use `let` as the default way to define a variable.

Generally, you should only ever have to use `let` and `const` to declare variables. You should use `const` by default, and if the value of the variable needs to be changed at some point, you can use `let` instead.

Fat Arrow Functions & Lexical Scope

One of my favourite new additions is fat arrow functions, which allow you to do something like this:

```
someFunction((response) => {
  console.log(response);
});
```

rather than:

```
someFunction(function(response){
  console.log(response);
});
```

At a glance, it might not seem all that great, but what this allows you to do is maintain the parent's scope. In the top example if I were to access the `this` keyword it would reference the parent, but in the bottom example I would need to do something like:

```
var me = this;

someFunction(function(response){
  console.log(me.someVariable);
});
```

to achieve the same result. With the new syntax, there is no need to create a static reference to `this` you can just use `this` directly. "Scope" is another concept that can be a little tricky to understand initially (scope refers to the context in which the code is being executed, and what it currently has access to). The main benefit of using this syntax in our applications is that we will always be able to easily access the methods/properties of our components, in effect, it kind of allows you to forget about dealing with "scope". When using this syntax, if we are writing code within **my-page.ts** then we know we will always be able to access any method or property we want in **my-page.ts** without having to worry

about scope.

There are very few circumstances where you wouldn't want to use a fat arrow function over a standard function, so if you ever find yourself writing `function` keyword in your application, go back and check to make sure you shouldn't be using a fat arrow function instead.

This is by no means an exhaustive list of new ES6 features so for some more examples take a look at [es6-features.org](). It is not super important that you understand all these concepts thoroughly right away, you should just be aware of them.

TypeScript

Another concept we should cover off on is TypeScript which is used in Ionic, so let's talk a little bit about what it is and how it is different to plain ES6. TypeScript's own website defines it as:

"a typed superset of JavaScript that compiles to plain JavaScript"

If you're anything like me then you still wouldn't know what TypeScript is from that description. The fact that TypeScript is a "superset" means that it still contains all of the normal JavaScript functionality, and you can actually just use "normal" Javascript in a TypeScript project, but it also adds some extra features. The most notable feature that TypeScript provides over standard ES6 is **types**.

Using TypeScript allows you to program in the way you would for stricter, object-oriented languages like Java or C# where variables are more strictly defined as a particular "type" like a string or a number. For example:

```
function add(x: number, y :number): number {
    return x + y;
}
add('a', 'b'); // compiler error
```

The code above states that `x` should be a number (`x: number`), `y` should be a number (`y: number`), and that the `add` function should return a value that is a number (`add(): number`). So in this example, we will receive an error because we're trying to supply characters to a function that expects only numbers. This can be very useful when creating complex applications and adds an extra layer of checks that will prevent bugs in your application.

If we take a look at some typical Ionic code:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  private myName: string = "Josh"
```

```
  constructor(private modalCtrl: ModalController){

  }

}
```

You can see some TypeScript action going on. The code above is saying that `myName` must have a type of `string`, and `modalCtrl` has a type of `ModalController`. As you will see later, the ability to give things types comes in very handy for an important concept called **dependency injection** (which we are using here to "inject" the `ModalController` into this class).

You can use types as much or as little as you like. If you don't like them, you can leave types off completely (except for dependency injection). However, the more you use types, the better the quality of your application will be. Supplying types allows the TypeScript compiler to catch errors before you ever even run into them, and if you are using an IDE like Visual Studio Code that supports TypeScript, it also supplies you with a lot of useful auto-completion information (e.g. if you've defined a method in another file that is expecting a "string" and a "number", your IDE will be able to tell you this, saving you having to go look it up yourself).

Transpiling

Transpiling means converting from one language to another language. Why is this important to us? Basically, ECMAScript 6 gives us all of this cool new stuff to use, but ES6 is just a standard and it is not completely supported by browsers yet. We use a transpiler

to convert our ES6 code into ES5 code (i.e. the Javascript you're using today) that *is* compatible with browsers.

In the context of Ionic applications, here's how the process works:

- You use `ionic serve` to run the application
- All the code inside of the **app** folder is **transpiled** into valid ES5 code
- A single bundled Javascript file is created and run

You don't need to worry about this process as it is all handled automatically.

Ionic and Angular Syntax

Now let's take a look at some actual Angular syntax that you will be using in your Ionic applications. Before I get into that though, I think it's useful to know about the APIs that each and every DOM element (that is, a single node in your HTML like `<button>`) have.

Let's imagine we've grabbed a single node by using something like `getElementById('myInput')` in JavaScript. That node will have **attributes**, **properties**, **methods** and **events**.

NOTE: We would not typically grab a reference to an element in an Ionic/Angular application using `getElementById`. This is just a generic example of how these concepts apply outside of the Ionic/Angular framework environment.

An **attribute** is some data you supply to the element, like this:

```
<input id="myInput" value="Hey there">
```

This attribute is used to set an initial value property on the element. Attributes can only ever be strings.

A **property** is much like an attribute, except that we can access it as an object and we can modify it after it has been created. For example:

```
var myInput = document.getElementById('myInput');
console.log(myInput.value); // Hey there
myInput.value = "What's up?";
console.log(myInput.value); // What's up?
myInput.value = new Object(); // We can also store objects instead
of strings on it
```

A **method** is a function that we can call on the element, like this:

```
myInput.setValue('Hello');
```

An element can also fire events like `focus`, `blur`, `click` and so on - elements can also fire custom events. Having a basic understanding of these native JavaScript concepts should help to shape your understanding of what Angular adds on top of that.

Ok, let's take a look at some Angular code! We will just be looking at a few quick examples, and we will dive into concepts in more depth later.

Property Binding

```
<input [value]="firstName">
```

This will set the elements `value` property to the expression **firstName** (which will be defined in the class definition for the component as a member variable). Note that `firstName` is an expression, not a string. Without the use of the square brackets, `firstName` would be treated literally as the string "firstName" rather than whatever it is that `firstName` evaluates to (like "Josh"). Surrounding the property with square brackets allows us to bind the value to a variable.

Calling a Function on an Event

```
<ion-button (click)="someFunction($event)"></ion-button>
```

This will call the `someFunction` function and pass in the event details whenever the button is clicked. You can replace click with any native or custom event you like. The `someFunction` function will need to be defined in the associated class definition for this template.

Interpolations

```
<p>Hi, {{ name }}</p>
```

Interpolations will evaluate the expression contained within the curly braces and render the result. If there is a variable called `name` contained in the associated class for this template, that name will be injected here (e.g. "Hi, Josh").

Two Way Data Binding

When we are creating a page or any kind of component, we will have a template that creates the view, and a class that holds all of our variables, functions, and so on. We might define data in the class, and use that data in the template.

Two-way data binding is a way to keep values in the template and in the class in sync. This means that if we changed a value in an input field it would immediately be updated in the class, and if we changed it in the class it would immediately be reflected in the template.

We can set up two way data binding in Angular like this:

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

This sets the value to the expression `name` and when we detect the input event we update `name` to be the new value that was entered. To make this process easier, we can use something called `ngModel` to achieve the same result:

```
<input [(ngModel)]="name">
```

which is just a shortcut for the above, but is obviously a lot easier to write. This syntax is basically saying "if the `name` variable is updated I want the value of the input set to that updated value immediately, and if my input is changed by the user I want the `name` variable to be updated immediately".

Creating a Local Variable

```
<p #myParagraph></p>
```

This creates a local variable that we can use to access the element, so if I wanted to add some content to this paragraph I could do the following:

```
<ion-button (click)="myParagraph.innerHTML = 'Once upon a
time...'"></ion-button>
```

There are also other ways you would be able to grab a reference to `myParagraph` but we won't get into that just yet. It would also be more common to make use of this local variable by referencing it in your class, rather than adding functionality directly to the `(click)` event as I have done in this example.

Embedded Templates

```
<p *something="someExpression"></p>
```

This star syntax allows you to create an embedded template, which would turn into this:

```
<template [something]="someExpression">
    <p></p>
</template>
```

Angular treats templates as chunks of the DOM that you can do stuff dynamically with. You will see this concept used a lot, but it isn't really important to understand the role of templates - in the **Structural Directives** section you are about to read this concept is used, but you don't need to know what is happening with templates behind the scenes in order to use it. The only time you would really need to worry about the role of templates is if you were building your own advanced custom directives.

Structural Directives

```
<section *ngIf="showSection"></section>
```

```
<li *ngFor="let item of items"></li>
```

Directives allow you to execute some behaviour on an element, and structural directives specifically allow you to affect the structure of the DOM. In the first example, we are conditionally showing the `<section>` based on whether or not `showSection` evaluates to be `true`, and in the second example, we are looping over all items in the items array and creating a list item for all of them.

These specific examples are "structural directives" because they use the `*` syntax and modify the DOM with templates. There are other kinds of directives that can be used to perform other tasks - you could have a directive that you attach to an element to set it to a random colour. This affects the behaviour of the element, but it doesn't modify the structure of the document. *Technically* speaking, all Angular components are directives (and an Angular component is just a directive with a template)... but it's really not important to get into the nitty-gritty of technical definitions right now.

Decorators

```
@Component({
  selector: 'my-component'
})
```

Decorators like **@Component**, **@Pipe**, and so on allow you to attach information to your components. This is a pretty important concept and we are going to spend a lot more time discussing it in its own lesson.

Import & Export

ES6 and TypeScript allows us to `import` and `export` components. Take the following component for example:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor(private dataService: DataService){

  }

}
```

This component is making use of `Component` from the Angular library and a data service that has been created elsewhere in the application, so it **imports** them here. The `HomePage` component that is being created here is then **exported**:

```
export class HomePage
```

which would allow it to be imported into another component in a similar fashion, i.e:

```
import { HomePage } from '../home/home';
```

Dependency Injection

Dependency injection is an important concept to understand. In most cases, if we are importing some service that we want to use we will also need to inject it into our constructor in order to make it available to the class. We can pass these services through our constructor as parameters, and then set up variables for them so that we can access them from anywhere in the class (rather than just in the constructor itself).

Since we can use **types** in TypeScript it is really easy for us to let our `constructor` know what we are trying to inject. Take a look at the following code for example:

```
import { Component } from '@angular/core';
import { Platform, ModalController } from '@ionic/angular';

@Component({
  selector: 'app-location-page',
  templateUrl: 'location.html',
  styleUrls: ['location.page.scss'],
})
export class LocationPage {
```

```
  constructor(modalCtrl: ModalController, platform: Platform) {

  }

}
```

We have given the first parameter a type of **ModalController** and the second parameter a type of **Platform**. So now our `constructor` knows that we want `modalCtrl` to be an instance of `ModalController` and `platform` to be an instance of `Platform`.

We can take this one step further by also adding in the `private` (or `public`) keyword like this:

```
import { Component } from '@angular/core';
import { Platform, ModalController } from '@ionic/angular';

@Component({
  selector: 'app-location-page',
  templateUrl: 'location.html',
  styleUrls: ['location.page.scss'],
})
export class LocationPage {

  constructor(private modalCtrl: ModalController, private
platform: Platform) {

  }
```

```
}
```

This automatically sets up a member variable for these values, which means you will be able to access them from anywhere within the class by referencing `this.modalCtrl` and `this.platform`. If something is not a "member variable" or "class member" then the variable is only accessible within the scope it was declared in (e.g. if you define a variable inside of a function, you can only access it inside of that function).

As I mentioned, this lesson was meant to give you a brief overview of the new concepts and syntax, rather than diving into details about everything. Hopefully, now you should have enough background to make learning Ionic a little bit easier.

Native Builds and Functionality with Capacitor

When we are building mobile applications with Ionic, we are building web applications. Since we are building web applications, our applications can run anywhere the web does. Obviously, this means we could put it on the World Wide Web without much trouble, and we could access through a web browser (on desktop, tablet, or phone). However, when it comes to mobile applications there are a couple of features that native mobile applications provide that web applications don't, like:

1. The ability to distribute the application through app stores
2. The ability to access Native APIs of the device (like Contacts, Bluetooth, or WiFi)

This is where **Capacitor** comes into play. Capacitor allows you to wrap your web based application in a native shell, and acts as a bridge to the native device. Capacitor will create a native application for each platform you are targeting (e.g. iOS and Android), it will set up a web view in that application, and then it will load your web application into that web view. The end result is a native application (that looks no different to any other native mobile application) that is running your web code.

I also mentioned that it acts as a "bridge to the native device". Since your code is running inside of a web view, your code does not have direct access to the devices Native APIs like a natively coded mobile application would. To get around this, Capacitor acts as a

"bridge" between your code and the native device. By using Capacitor APIs or plugins, you can make a request to Capacitor for certain functionality. Capacitor will then forward your request to the native device, and pass the result back to your web code.

If you are already familiar with Cordova/PhoneGap, then you can consider Capacitor to fulfill a similar role. Capacitor also supports most existing Cordova plugins, so you can use Cordova plugins inside of a Capacitor project.

I think that the Capacitor website describes its goals quite succinctly: **Native Web Apps**. Capacitor wants to be a runtime layer that allows you to build an application with web tech and have it run anywhere (even if you are integrating native code). The role of Capacitor isn't just to allow you to build iOS and Android apps, it's to provide a consistent API that will allow you to easily share your code across all platforms.

The Role of Capacitor

I mentioned that Capacitor is similar in spirit to Cordova, and it is basically the Ionic teams own version of Cordova - built from the ground up with features they felt were necessary for pushing Ionic applications forward.

So, what kind of improvements are the Ionic team making? Here are a few things that stand out to me in particular:

PWA Compatibility. There has been a huge movement for **Progressive Web Applications** recently. A PWA, in brief, could be decribed as a web app that can be installed on your device (directly from the website itself, not an app store) and that works offline (but PWAs encompass a lot more than just that). They are starting to receive more support from browsers, and more tools are starting to pop up to help people build PWAs.

One cool thing about a PWA is that it can also easily be bundled as a native application and access native functionality (using Cordova/Capacitor). Native code will not work when running as a PWA through the browser, though. This can make maintaining a single codebase for a project that will be distributed as a native application and as a PWA (and perhaps even as a desktop application as well) difficult. With Capacitor as the runtime layer, it will be able to gracefully handle any errors that arise from attempting to access native functionality in an environment that does not have it. Capacitor is also aiming to make many of these APIs work on the web also (like Camera support for the web).

Ease of integrating native controls. Capacitor is aiming to make it easier to include native user interface controls wherever desired. For example, if you wanted to use a native menu rather than using the browser to create that interface. In general, Capacitor gives you a lot more control over the actual native application that is built.

Native functionality available immediately. When using Cordova, you need to wait until the device is ready before making calls to native functionality (e.g. by using `platform.ready()`). Capacitor will export JavaScript on app boot so that this is no longer required.

Plugin support. All native functionality is accessed through a plugin of some sort. If you want to use the Camera, then you use the camera plugin. If there isn't a plugin available to do what you want, or there is but it is not being maintained, you are somewhat out of luck unless you know how to build the plugin yourself. Capacitor is aiming to make the process of creating and maintaining plugins easier.

Redesigned plugins. Capacitor will include a core set of plugins (Camera, File API, Geolocation, etc.) that will be maintained by the Ionic team. They plan on redesigning these plugins to solve some common issues (like the difficulty of using the File API, and the

way Android Geolocation works).

Localised installation. Capacitor will be installed locally in projects, meaning that it will be easier to maintain multiple different versions between multiple projects.

Although Capacitor operates quite differently to Cordova, the Ionic team are still aiming for backward compatibility with as many Cordova plugins as possible. Since there is such a huge community already built up around Cordova, being able to use those plugins will be a great benefit.

If Cordova/Capacitor/Native bridges etc. are all new to you, then I've probably just thrown a lot of concepts at you that don't make much sense. If that is you, don't worry, it's not important to understand all the benefits Capacitor provides. All you need to know is that it will allow you to build your web apps as native apps, and provide access to native functionality.

Installing Capacitor

We will be installing Capacitor in an Ionic application later, but it is important to remember that Capacitor isn't just for Ionic applications. You can use Capacitor with pretty much any web based application. All that is required to use Capacitor is to install it into an existing project:

NOTE: Don't follow these steps just yet (unless you have some project you want to try Capacitor out on right now). We will use the Ionic CLI to set Capacitor up in our projects later.

```
npm install @capacitor/core @capacitor/cli
```

and then run:

```
npx cap init
```

The initialisation process will configure the basics. The main thing you need to configure is pointing Capacitor to the folder where your built web code lives so it knows what to wrap up into the native application. Once you have done this, you can start using Capacitor in your application.

There is a little more that needs to be done when you are ready to build your application and run it on a device, but we are going to get into that later.

Using Capacitor APIs

We are also going to cover using various aspects of Capacitor in more detail later, but I want you to have a little context now so that Capacitor isn't this abstract concept in your head. Here is an example of how you would use the Camera API from Capacitor in an Ionic application:

```
import { Plugins } from "@capacitor/core";
```

```
const { Camera } = Plugins;
```

To make the Camera API available in a file you need to first import `Plugins` from the `@capacitor/core` library, and then set up a reference to the Camera plugin. Then to use this reference, you would just do this:

```
try {
  const photo = await Camera.getPhoto();
} catch (err) {
  console.log("Could not get photo");
}
```

When this code is triggered, it will launch a prompt on the user's device asking them if they want to use their Camera or pick a photo from their library. The user will then be able to select or take a photo, and once they are done, the photo will be passed back into your application using the promise as shown above! In case of any errors that occur (like the user cancelling the dialog) we use a `try/catch` block to handle failures.

All Capacitor APIs work in more or less this format:

```
const result = await SomePlugin.someMethod();
```

Ionic Basics

The first section of this book is dedicated to learning basic concepts and syntax. If you like, you can follow along by generating an application and trying out some of the examples for yourself, but this section is more about just getting you somewhat familiar with everything in a broad sense. We will focus on coding through guided examples a little later in the book.

Lesson 1: Generating an Ionic Application

We've covered quite a bit of context already, so hopefully, at this point, you should be starting to get somewhat comfortable with some of the Ionic/Angular specific syntax. With that in mind, we're ready to jump in and start learning how to actually use Ionic with Angular.

Setting up Your Environment

Before we can install and start building an application with the Ionic CLI we need to get everything set up on our computer first. The key things we will need to do are:

- Install the LTS version of NodeJS
- Install the Ionic CLI
- Install Xcode (if you want to build iOS applications locally) - *requires macOS*
- Install Android Studio (if you want to build Android applications locally)

Since we will be using Capacitor throughout this book, you may still need to update your

version of NodeJS even if you have it installed already.

If you don't want to bother with installing/setting up Xcode and Android studio now, you don't have to. This is only required for creating the native builds that will run on devices and can be submitted to the Apple App Store and Google Play. If you just want to jump in and start learning Ionic, this step can be done much later (and if you are just interested in creating a Progressive Web Application, you won't ever need to install Xcode or Android Studio).

An important thing to note is that macOS is required to install Xcode and create native builds with Capacitor. This could be your own Mac computer, one you borrow from a friend or even a virtual macOS machine that you might access through the Internet. There is no getting around the fact that the Mac operating system is required in some capacity to build native builds for iOS, though. Ionic does provide a paid service called Appflow that can perform the iOS builds for you (so that you don't need a Mac yourself), so that is an option if you don't have a Mac but want to build iOS applications. Appflow is also a great option in general for managing builds and deploys for both iOS and Android.

Installing Node

You will need to have Node installed on your machine in order to use Capacitor. I would recommend downloading the LTS version of NodeJS from the NodeJS website.

It is also a good idea to install the n package (unfortunately, n is not available for Windows - you do not need it, it's just a helpful tool to have). This is a version manager for Node, and allows you to easily switch to whatever version of Node you want to use. Capacitor requires at least Node version 8.6.0 for example, but other tools you use may require

different versions. Using `n` allows you to run a one-line command to switch between versions. To install `n` just run the following command (after installing NodeJS):

```
npm install -g n
```

Then you can switch to whatever Node version you like using the following command format:

```
n 14.15.1
```

As a default, you should use whatever the current **LTS** version is (as listed on the website).

Install the Ionic CLI (Command Line Interface)

Once you have NodeJS installed, you will be able to access the **node package manager** or **npm** through your command terminal. This will allow you to install the Ionic CLI.

Keep in mind that the Ionic framework and the Ionic CLI are two different things. You aren't actually installing the Ionic framework, just the CLI. After you have installed the CLI, you will use it to generate a project that includes the Ionic framework.

> Install the Ionic CLI by running the following command in your terminal:

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

or

`sudo npm install -g @ionic/cli` (on Windows you can run as administrator instead of using `sudo`).

To ensure that the Ionic CLI has been installed successfully, you can run the following command to check:

```
ionic --version
```

This should output the current version of the Ionic CLI that you have installed. If you are having trouble with your installation, you can always run the `doctor` command to attempt to check automatically for issues:

```
ionic doctor check
```

and you can use the following command to attempt to automatically fix issues:

```
ionic doctor treat
```

If you are still having trouble, a good step for flushing out issues is to completely uninstall the Ionic CLI:

```
npm uninstall -g @ionic/cli
```

and then try again after ensuring you have an appropriate version of NodeJS and npm installed.

Installing a Text Editor/IDE (Integrated Development Environment)

The beautiful thing about working with web tech is that in the end all you are doing is editing text files, so you can use whatever program you are most comfortable in to do that.

However, if you don't already have a preference (and maybe even if you do) I would strongly recommend checking out Visual Studio Code. It has excellent support for TypeScript and has auto code completion that can help you import from the correct path, and in general, it can just highlight issues before you even notice them. If you attempt to assign invalid data to a variable or attempt to access a method that doesn't exist, VS Code will warn you right there in the editor before you even run your code.

It can even help you learn various APIs that you will need to use in your projects, as the auto code completion will pop up suggestions that may save you from needing to look things up in the documentation.

Installing Xcode (Optional)

In order to run applications on your iOS device, and submit them to the Apple App Store, you will need Xcode. You can download this from the Apple App Store (this will require a Mac).

NOTE: You can perform this step later if you wish.

As well as installing Xcode itself, you will also need to install Cocoapods and the Xcode CLI tools. You can do that by running the following commands:

```
sudo gem install cocoapods
```

```
xcode-select --install
```

Once you have installed Xcode, make sure to open it at least once so that it can install the necessary components. You should also run the following command from time to time to make sure the CocoaPods dependencies are up to date (Pods are kind of like npm packages for iOS):

```
pod repo update
```

Installing Android Studio (Optional)

We will need to use Android Studio to build native Android applications and submit them to Google Play. If you do not already have Android Studio, you should install it by visiting this website.

NOTE: You can perform this step later if you wish.

Once it is installed, you should open it. The first time you open Android Studio you will be taken through a setup wizard:

Just select the **Standard** options. When asked to choose between the *light* side and the **dark** side, choose wisely:

The wizard will then handle installing all of the dependencies you will need, which includes things like the SDK (Software Development Kit) for development, an emulator, and build tools.

Hit **Finish**. If you need to install or update any components in the future, you can do so using the **SDK Manager** that comes with Android Studio by going to **Tools > SDK Manager**.

Generating Your First Project

Now that we have the environment set up, we can generate our first application!

Once Ionic is installed, generating applications is really easy. You can simply run the `ionic start` command to create a new application with all of the boilerplate code and files you need. Just using `ionic start` will then prompt us with additional options to

specify what kind of application we are creating (e.g. Angular, React, Vue). To save time, we can just supply some additional configuration to the `ionic start` command like we will do below.

> Run the following command to generate a new application:

```
ionic start MyFirstApp blank --type angular
```

There are other templates aside from `blank` that we can base our project on, but we will always use the `blank` template as we want to learn as we build, and this is also what I would generally use for projects to save from having to remove unnecessary boiler plate code. However, the starter templates other than `blank` can be very useful if they closely match the structure of the application that you want to build.

Once the project has finished generating, you will need to make it your working directory before you can serve it.

> Run the following command to change to the directory to your new Ionic project

```
cd MyFirstApp
```

Running the Application

The beauty of web-based mobile applications is that you can run them right in your browser whilst you are developing them. But if you try just opening up your project in a browser by going to the **index.html** file location you won't have a very good time.

An Ionic project needs to run on a web server - this means you can't just run it by accessing the file directly, but it doesn't mean that you actually need to run it on a server on the Internet, you can deploy a completely self contained Ionic app to the app stores (which we will be doing). Fortunately, Ionic provides an easy way to view the application through a local web server whilst developing.

> To view your application through the web browser run the following command:

```
ionic serve
```

This will open up a new browser with your application open in it and running on a local web server. Right now, it should look something like this:

Not only will this let you view your application but it will also update live with any code changes. If you edit and save any files, the change will be reflected in the browser without having to reload the application by refreshing the page.

To stop this process just hit:

```
Ctrl + C
```

when you have your command terminal open. Also keep in mind that you can't run normal

Ionic CLI commands whilst `ionic serve` is running, so you will need to press Control + C before running any commands.

Updating Your Application

There may come a time when you want to update to a newer version of Ionic. Perhaps the easiest way to update the version of Ionic that your application is using is to first update the Ionic CLI by running:

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

or

```
sudo npm install -g @ionic/cli
```

Once you have the latest CLI installed, you will be able to **create a new project**. This new project will have all of the latest dependencies listed in its **package.json** file. If you update the dependencies in your current applications **package.json** file to match the dependencies in the new **package.json** file, and then run `npm install` in your old project, your application should be updated to the latest version. Make sure that you don't remove any additional dependencies that you have added to your project when performing this process.

Make sure to read the changelog to check for any breaking changes when a new version is released, as sometimes you may need to update parts of your code after updating to newer versions.

Lesson 2: Anatomy of An Ionic Project

Now that we've covered how to get Ionic installed and how to generate a project, I want to cover what the various files and folders contained within your newly generated project do. When you create a blank Ionic application, your folder structure will look like this:

At first glance, there is an intimidating amount of stuff there - but you don't need to know

what *everything* does. In this lesson, we are going to talk through the files and folders that you will be using most frequently, and what they are responsible for. We won't be covering *every* file and folder in the project - just enough to give you a decent overview of how everything fits together.

src

Perhaps the most important folder in your project is the **src** folder. This folder contains most of the files and folders that you will actually be working on to create your application. There are other files and folders outside of this, but you generally won't need to touch those (much).

app

You will find the **app** folder inside of the **src** folder, and this folder holds a similar level of importance. Most of the code for your application will live inside of this **app** folder. The **app** folder will contain several other folders like the **services** folder that will hold any services. As well as these folders that will contain various parts of your application, you will also find several `app` files in the root of this **app** folder. These files are very important to your application, so we are going to talk through what these files do:

- `app.module.ts`

Module files use an Angular concept called `@NgModule`. It is not important that you understand the intricacies of `@NgModule`, but the basic idea is that modules allow an application to be split up into "chunks" of related functionality. A "chunk" of functionality might be a page, or the various things required to make forms work, or all the functionality

required to make data storage work. You will use some modules that are created for you (like the `FormsModule` that allows for form usage, or the `IonicStorageModule` that allows for data storage), and you will also create some of your own modules (mostly this will just be modules for each of your pages).

You will have multiple modules throughout your application (e.g. for each of your pages) but the module found in **app.module.ts** is the **root** module for your application. The "chunk" of functionality that this module imports will be used by the entire application. Module files used by individual pages are only importing functionality for use by that particular page. This ties into the concept of "lazy loading" that we will discuss later, where can save time by only loading in the chunks/modules that we *need* at any given time. We will always need to load the root module, but we don't necessarily need to load the other modules in the application (until they are accessed).

If you were to inspect this particular file, you will see that it imports various other modules like the `BrowserModule` and `IonicModule`. These modules each then include all of the dependencies they need. By importing a single `IonicModule`, we get all the functionality that we will require from Ionic into our application (opposed to having to import a million different things ourselves). You generally won't need to touch this file, but you may need to on occasion to import additional functionality. If you wanted to make HTTP requests in your application you would need to import the `HttpClientModule`, or if you wanted to use Ionic's Storage API you would need to import the `IonicStorageModule` in your root module file.

- `app-routing.module.ts`

Another module file! This file creates a module called `AppRoutingModule` which is then

imported into the root module mentioned above. The purpose of this module is to define the routes used in the application. We will be discussing this in a lot more depth later, but the basic idea is that we want to define what URL matches up to which component/page. If the user goes to the `/home` route then we will (usually) want to display the home page/component.

This doesn't *need* to be in its own module, you could just set these routes up directly in the main root module. However, it is more organised to separate this out into its own file, and this kind of modularisation and separation of concerns is pretty much the point of modules.

- `app.component.ts`

This just defines a regular Angular component, which is what all the pages in our applications will be, but the important thing about this is that it is the **root** component. All other components/pages we create the application will be added *inside* of this component.

- app.component.html

This file contains the template for the root component, and you will see that all it contains is the following:

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

The `<ion-router-outlet>` is what controls what component is displayed to the user based on the current route. If the user were on the `/home` route, then the home page would be displayed in here. That home page component might also have more components that live inside of that. We will discuss this stuff in a lot more detail, but perhaps you can see how we are creating this tree or nest of components that is our application, where the **root** component is at the base of that tree (or that the **root** component surrounds that nest).

pages

By default, pages (like the `home` page) will be found inside of the `app` folder. A "page" is no different to any other Angular component, it's just that these particular components are the ones that we are using to display each "screen" in our application - like the home page, login page, product detail page, and so on. We might have other components in the application like a rating component that displays a 5-star rating - this is no different to our "page" components in the way it is coded, but conceptually we would be using components like this for a different purpose and so we separate them out into their own **components** folder. You will not find a **components** folder by default in your applications - at a beginner level, you won't often create custom components apart from the components you are using for pages.

A page/component is made up primarily of:

- A `X.module.ts` file which imports required functionality
- A `X.page.ts` which defines logic/behaviour related to the page
- A `X.page.html` which defines the template/view for the page
- A `X.page.scss` file which defines the styling for the page

We will, of course, be covering all of this in a lot more detail. For now, I just want to give you a broad overview.

services

This folder will contain providers/services/injectables that your application will make use of. A "service" can be any number of things, but in general, services are used to "do work" that will be used by the components in your application. You might have a `DataService` that is responsible for loading and saving data or a `UserService` that is responsible for keeping track of the logged in user's data.

The benefit of using services is that it abstracts potentially complex logic/code out from the components, and services can also be shared and used by multiple different pages/components at the same time. We will be discussing services in a lot more detail throughout the book.

assets

This folder is used to store any static assets for your application. This typically means images that you want to bundle with your application, but you can include any kind of static asset you like in here.

theme

The theme folder holds the `variables.scss` file for your application, which is where you can define CSS4 variables to modify the theme of your application (more on this later).

www

You won't ever need to touch this folder, but it is important that you understand what it does. As I mentioned in previous lessons, the code we write for our application is "transpiled" into code that is understood by browsers. The **www** folder contains the output of the build process for the application, and the code contained in this folder is the code that is actually run when a user uses the application. It is important not to make changes to the code in this folder, because it will be overwritten every time a new build is created.

If you were deploying your application to the web, it is the code in this folder that would be uploaded to the server hosting the application. If you are deploying your application to iOS or Android with Capacitor, it is the code in this folder that Capacitor will wrap up into the native shell.

package.json

This file contains configuration information for your project. The most important thing to understand is that it lists all of the dependencies (external libraries/packages required to make your project work) for your application. You will see multiple dependencies listed in here by default, mostly referencing packages from Ionic or Angular.

When a project is created, the `npm install` command is run, which installs all of these dependencies in your project. All of these dependencies are added to the **node_modules** folder. From there, these dependencies can be imported into your project. For example, we will frequently use this:

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

Which is importing from the `@angular/core` package (one of the dependencies listed in the **package.json** file). Although it is not always required, you can also install additional dependencies in your application (generally if you want to integrate some 3rd party library to provide additional functionality). To do that, you would run a command that looks like this:

```
npm install some-package
```

This would also add the dependency to your **package.json** file. The reason we want all of our dependencies listed in **package.json** is because we can then run `npm install` at any time to make sure we have all of the required dependencies installed for the application. This is especially important if you are using a service like GitHub to manage your applications codebase - our dependencies aren't uploaded to GitHub, and accurate dependencies listed in the `package.json` file will allow us to simple run `npm install` after cloning a project from GitHub to install all of the required dependencies.

Summary

As I mentioned, what we have discussed here is not *all* of the files in your project, but these are the key ones that you will need to worry about.

Lesson 3: Ionic CLI Commands

The Ionic CLI is a super powerful tool - we've already gone through how to use it to generate a new project and display your application in the browser, but there are a bunch more commands you should know about too, so let's go through some of them. This is by no means an exhaustive list, but it will cover all of the commands you should be using frequently.

Let's get into it!

Serving your Application

As you probably already know, you can view the Ionic application that you are working on in the browser by running:

```
ionic serve
```

However, you can also serve your project using **Ionic Lab** which will display your project in the Ionic Lab interface. This will give you a side-by-side comparison of what your application will look like on both iOS and Android, and there are also some useful links on the left-hand side of the screen.

If you would like to use Ionic Lab, you will first need to install the package for it in your project:

```
npm install @ionic/lab
```

and then you can add the `-l` flag to the `serve` command:

```
ionic serve -l
```

Generate

The generate command is one of my favourites, it's a real time saver. When you are developing an application you will eventually want to add more pages/components/services than the default ones that are created with the application. To create more components you can manually create a new folder and add all the required files, or you can just run the `ionic generate` command to do it automatically for you, with some handy boilerplate code in place.

You can use the generate command to automatically create a:

- page
- component
- directive
- service

To use the command you can just run:

```
ionic g page
```

and answer the prompt, or you can manually supply the name of your page/component/directive/service to the command like this:

```
ionic g page MyPage
```

or

```
ionic g service services/MyService
```

If you are running the manual command, make sure to supply the folder you want it to be generated in (e.g. as we have done for the service above) otherwise it will just be generated in the root folder of your application code.

Installing Packages or Plugins

You will often want to install 3rd party packages or plugins when building an application. Most of the time, the instructions will say to run a command like this:

```
npm install some-package
```

That is fine, but for some plugins, you may be instructed to add them like this:

```
ionic cordova plugin add some-plugin
```

or this:

```
cordova plugin add some-plugin
```

DO NOT do this if you are using Capacitor. You should only run those commands if you are using Cordova (we will be using Capacitor in this book). Instead, install the plugin directly with npm, e.g:

```
npm install some-plugin
```

If you want to use Ionic Native to access the plugin (assuming that it is available), you should still install the Ionic Native package as well:

```
npm install @ionic-native/some-plugin
```

When importing plugins from Ionic Native into your project, you should also import from `/ngx` e.g:

```
import { Facebook } from '@ionic-native/facebook/ngx';
```

Creating a Build

Most of the time we will just use `serve` to run our application, but sometimes you will want to create a build of your application in the `www` folder that you can access directly. You would want to do this perhaps if you wanted to run your code on the web somewhere, or if you are creating native builds with Capacitor you will also need to create a build of your project (since Capacitor will copy the source code in the `www` folder to your native projects).

To create a build, you can just run the following command:

```
ionic build
```

or to create an optimised production build you can run:

```
ionic build --prod
```

Using Capacitor

Although Capacitor and the Capacitor CLI can be used on its own in any project, the Ionic CLI also has integrations for Capacitor. To initialise Capacitor in your Ionic project you can just add a platform:

```
ionic cap add android
```

or

```
ionic cap add ios
```

or both. This will save you the hassle of having to install Capacitor and initialise your project, but it will give your project a default Bundle ID of `io.ionic.starter`. This is fine for development, but you will want to change it to something unique (e.g. `com.yourcompany.projectname`) before submitting your project. You can change this in the **capacitor.config.json** file, and you will also need to make sure that it is updated in your Xcode/Android Studio projects.

If you try to run the `add` commands before running `ionic build` you will get an error, but that's fine, just make sure you do a build at some point. Once you have created a build, you can run the following command:

```
ionic cap sync
```

To sync your project with Capacitor. This will copy over your built application from the `www` folder, it will update the dependencies in your native projects, and it will install any plugins that you have installed in your project.

IMPORTANT: Running `ionic cap sync` will also automatically create a build of your project, which will be in development mode by default. If you want to deploy a production build with Capacitor you should run the command with the `--prod` flag: `ionic cap sync --prod`.

You can also run:

```
ionic cap update
```

which will only update the native dependencies/plugins, or you can run:

```
ionic cap copy
```

which will only copy your application code over to the native projects. The `sync` command just does both of these at once. To open your project in Xcode you can run:

```
ionic cap open ios
```

and you can also run your application on an emulator or on a real device from there. There is more details later in the book on exactly how to do this. The same goes for Android, you can run:

```
ionic cap open android
```

to open your Android project in Android Studio, and you will be able to run your application from there. Unlike with Cordova, all of the building and running steps for the native project in Capacitor are handled by their respective native tools.

Doctor

We've made it through all of the commands that you would likely use on a day-to-day basis, but there are a few more commands that are useful to know if you get stuck. Sometimes you can run into problems with the way your environment is configured, especially when just getting started, and these issues can be frustrating to solve. To help with that, you can run the following command:

```
ionic doctor check
```

This will run some diagnostics to check for any potential issues. You can even run this command:

```
ionic doctor treat
```

to attempt to automatically fix issues.

Help

Finally, the `--help` flag can be extremely useful if you are unsure how certain commands work. You can basically add `--help` to any command to get details on how that command works. For example, you could run:

```
ionic —-help
```

or more specifically, commands like:

```
ionic cap —-help
```

or:

```
```

```
ionic g --help
```

We haven't covered every single command in this lesson, but these are likely the only commands that the Ionic CLI provides that you will use frequently.

Lesson 4: Decorators

Each class (which we will talk about in the next section) you see in an Ionic application will have a **decorator**. A decorator looks like this:

```
@Component({
    someThing: 'somevalue',
    someOtherThing: [Some, Other, Values]
})
```

They definitely look a little weird, but they play an important role. Their role in an Ionic application is to provide some *metadata* about the class you are defining, and they always sit directly above your class definition (again, we'll get to that shortly) like this:

```
@Decorator({
    /*meta data goes here*/
})
export class MyClass {
    /*class stuff goes here*/
}
```

The `@Component` is just one type of decorator, but there are other types as well. I have

used `@Decorator` above as a generic example, but keep in mind that there isn't a decorator literally called `@Decorator` (I'm just using it as a placeholder).

This is the only place you will see a decorator, they are used purely to add some extra information to a class (i.e. they "decorate" the class). So, let's talk about exactly how and why we would want to use these decorators in an Ionic application.

The decorator name itself is quite useful, here are a few you might see in an Ionic application:

- `@Component`
- `@Pipe`
- `@Directive`
- `@Injectable`

Just the decorator itself is enough to prove extra information about the class:

```
@Injectable()
export class MyService {

}
```

But we can also supply an object to the decorator to provide more information about the class. Here's the most common example you'll see in your applications:

```
@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

}
```

With the help of the decorator, this class is identified as a component and that:

- The tag for the component will look like this: `<app-page-home></app-page-home>`
- It needs to fetch its template from `home.page.html`, which will determine what the user will actually see on the screen (we'll be getting into that later as well).
- It needs to fetch the styles for this component from `home.page.scss`

These are the most commonly used properties for the `@Component` decorator, but there are more. If you've got a super simple template, maybe you don't even want to have an external template file, and instead define your template like this:

```
@Component({
    selector: 'app-page-home',
    template: `<p>Howdy!</p>`,
    styleUrls: ['home.page.scss']
})
```

```
export class HomePage {

}
```

Some people even like to define large templates using the `template` property. Since ES6 supports using backticks (the things surrounding the template above) to define multi-line strings, it makes defining large templates like this a viable option if you prefer (rather than doing something ugly like concatenating a bunch of strings).

Now that we've covered the basics of what a decorator is and what it does, let's take a look at some specifics.

Common Decorators in Ionic Applications

There are quite a few different decorators that we can use. In the end, their main purpose is simply to describe *what* the class we are creating *is* so that it knows what needs to be imported to make it work.

Let's discuss the main decorators you are likely to use, and what the role of each one is. We're just going to be focusing on the role of the decorator, for now, we will get into how to actually build something useable by defining the class in the next section.

@Component

I think the terminology of a *component* can be a little confusing in Ionic. As I mentioned, our application is made up of a bunch of components that are all tied together. These

components are contained within folders inside of our main **src** folder, and they look like this:

home

- `home.page.ts`
- `home.page.html`
- `home.page.scss`

NOTE: Depending on how your application is set up, your pages may consist of more files than just these three. However, these three files are the key parts of creating a component.

A **@Component** is not specific to Ionic, it is used generally in Angular. A lot of the functionality provided by Ionic is done through using components. In Ionic, for example, you might want to create a search bar, which you could do by using one of the components that Ionic provides like this:

```
<ion-searchbar></ion-searchbar>
```

You simply add this custom tag to your template. Ionic provides a lot of components but you can also create your own custom components, and the decorator for that might look something like this:

```
@Component({
```

```
    selector: 'my-cool-component'
})
```

which would then allow you to use it in your templates like this:

```
<my-cool-component></my-cool-component>
```

The `@Component` decorator will likely be the one you use the most. Even if we aren't creating our own custom components like lists and buttons, each page in your application is still its own component. Always keep in mind that your whole application is just components within components within components. Your entire application is a component, and inside of that component you have **pages** for your application which are also components, and inside of those components, you might use even more components like Ionic's buttons and lists.

NOTE: Technically speaking a component should have a class definition and a template. Things like pipes and services/providers aren't viewed on the screen so have no associated template, they just provide some additional functionality. Even though these are not technically components you may often see them referred to as such, or they may also be referred to as services or providers.

@Directive

The **@Directive** decorator allows you to create your own custom directives. We've

touched on the concept of a directive briefly, but basically, it allows you to attach some behaviour to a particular component/element. Typically, the decorator would look something like this:

```
@Directive({
    selector: '[my-selector]'
})
```

Then in your template, you could use that selector to trigger the behaviour of the directive you have created by adding it to an element:

```
<some-element my-selector></some-element>
```

It might be a little confusing as to when to use **@Component** and **@Directive**, as they are both quite similar. The easiest thing to remember is that if you want to modify the behaviour of an existing component use a **directive**, if you want to create a completely new element/component use a **component**. As I mentioned before, technically speaking, a component is still considered to be a directive (it is just a directive with its own template, rather than a directive that attaches to another component).

@Pipe

@Pipe allows you to create your own custom pipes to filter data that is displayed to the

user, which can be very useful. The decorator might look something like this:

```
@Pipe({
  name: 'myPipe'
})
```

which would then allow you to implement it in your templates like this:

```
<p>{{someString | myPipe}}</p>
```

Now `someString` would be run through your custom `myPipe` before the value is output to the user. The main role of a pipe is to take some data in (`someString` in this case) and then return that data in a different format. A common example would be doing things like converting the format of currencies or dates to display in a more friendly manner (e.g. 13th of April, 2018 rather than 2018/04/13).

@Injectable

An **@Injectable** allows you to create a service for a class to use. A common example of a service created using the **@Injectable** decorator, and one we will be using a lot when we get into actually building the apps is a **Data Service** that is responsible for fetching and saving data. Rather than doing this manually in the classes for your pages, you can inject your data service into any number of classes you want, and call helper functions from that

Data Service. Of course, this isn't all you can do, you can create a service to do anything you like.

An **@Injectable** will often just look like a normal class with the **@Injectable** decorator tacked on at the top:

```
@Injectable({
  providedIn: 'root'
})
export class DataService {

}
```

In older versions of Angular, we would need to add our `@Injectable` classes to the `providers` array in the root module of the application. We can now use this `providedIn` property to make it available to use in our application. By providing the service in `root` it will be provided to the root module of the application, which will allow it to be used as a "singleton" throughout the entire application. A singleton means that one instance of the class will be instantiated for use through the entire application - so if you access the service from two different pages, you are going to be accessing the same data. If you set some variable in the service on `PageOne`, you could then grab that variable from `PageTwo` since it is a singleton. This is typically how you would use services in your application.

Summary

The important thing to remember about decorators is: *there's not that much to remember*.

Decorators are powerful, and you can certainly come up with some complex looking configurations. Your decorators may become complex as you learn more about Ionic, but in the beginning, the vast majority of your decorators will probably just look like this:

```
@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

}
```

I think a lot of people find decorators off-putting because at a glance they look pretty weird, but they look way scarier than they actually are. In the next lesson, we'll be looking at the decorator's partner in crime: the class. The class definition is where we will do all the actual work, remember that the decorator just sits at the top and provides a little extra information.

Lesson 5: Classes

In the last section, we covered what a **decorator** is. To recap, it's a little bit of code that sits above our class definitions that declares *what* the class is, and how the class should be configured. Now we are going to talk about the **class** itself.

What is a Class?

Depending on your experience with programming languages, you may or may not know what a **class** is. So I'm going to take a step back first and explain what a class is as a general programming concept, as it is not specific to Ionic, Angular, or even Javascript.

Classes are used in Object Oriented Programming (OOP), they essentially behave as "blueprints" for objects. You can define a class, and then using that class you can create, or "instantiate", objects from it. If classes are a completely new concept to you, it'd be worth doing a little bit of your own research before continuing, but let's take a look at a simple example.

```
class Person {

  constructor(name, age){
    this.name = name;
    this.age = age;
  }
```

```
  setAge(age){
    this.age = age;
    return true;
  }

  getAge(){
    return this.age;
  }

  setName(name){
    this.name = name;
    return true;
  }

  getName(){
    return this.name
  }

  isOldEnough(){
    return this.age > 70;
  }
}
```

This class defines a **Person** object. The `constructor` is run whenever we create an instance of this class (an object is an instance of a class), and it takes in two values: name and age. These values are used to set the **member variables** or **class members** of the class, which are `this.name` and `this.age`.

These values can be accessed from anywhere within the object by using the `this` keyword. The `this` keyword refers to the current **scope** of wherever `this` is used, so what it evaluates to depends on where you use it, but if you use it within a class (and not within a callback or anything else which would change the scope) this will refer to the class/object itself.

If you imagine yourself as `this` and your location in the physical world as the **scope**, consider the following example: if you are in a house, your "scope" may include the room you are in, all of the areas of the house, and even the entire world. These are areas you are free to explore and access. In a programming sense, if you are inside of a function you can access things inside of that function (your room), other properties/methods of the class that contains that function (other rooms/items in the house), or the entire global scope of the application (the world around the house). When you are anywhere inside of the house, `this` would be a reference to the house (except in special circumstances). Although you can access anything inside of your own house, you can not access things in other peoples houses (unless you have permission to do so).

In the analogy above, we could use the `this` keyword to access other methods/variables of the class (rooms/items of the house). If we were inside of one function, but want to access another in the same class, we would do something like: `this.someOtherFunction()`.

If you're not familiar with the `this` keyword, I'd recommend reading this. It is a tricky concept to get your head around, fortunately, this is one of those things that isn't critical to building applications with Ionic. Context and understanding always help you improve as a developer, but it's not going to stop you from progressing. In general, you will only use `this` when you need to refer to other methods or variables of the class that you are in. But

it is important to understand that you can't just freely access variables and methods of *other* classes.

Once we have our class defined which acts as a blueprint for creating objects, we could create a new **Person** object like this:

```
const john = new Person('John', 32);
```

The two values I've supplied here will be passed into the `constructor` of the **Person** class to set up the **member variables**. Now if I were to run the following code:

```
console.log(john.getName());
```

John's name would be logged to the console. Similarly, we could also call the `getAge` function to retrieve his age or we could even change his name or age using the set functions. Getters and setters are very common for classes, but we've also defined a more interesting function here which is `isOldEnough`. This will return true if the **Person** is over 70 years old, which is not the case for John.

Perhaps the most important concept to remember is that the class is just a "blueprint", an object is kind of like an individual copy of a class. So we can have multiple objects created from the same class, e.g:

```
const john = new Person('John', 32);
const louise = new Person('Louise', 28);
const david = new Person('David', 72);

console.log(john.isOldEnough());
console.log(louise.isOldEnough());
console.log(david.isOldEnough());
```

In the code above, John, Louise, and David are all individual objects of the **Person** class and maintain their values separately. If we ran the code above, it would only return **true** for David.

Classes in Ionic

Why do we need classes for Ionic/Angular applications? We've touched on this earlier, but **classes** are a new addition to Javascript with the ES6 specification. It is certainly a welcome change because it is one of the most widely used patterns in programming, and in fact, most JavaScript applications were already using class structures anyway, ES6 just made it more official.

Before ES6 it was common (and I guess it still is since most people are still using ES5) to create a class like structure by using **functions**. This looks a little something like this:

```
var Person = function (name, age) {
  this.name = name;
  this.age = age;
```

```
};

Person.prototype.isOldEnough = function() {
  return this.age > 70;
};

var david = new Person('david', 72);
console.log(david.isOldEnough());
```

It looks a bit different, but the end result is basically the same. Since ES6 adds a class keyword we can now use a more 'normal' approach.

So, let's take a look at what a class looks like in Ionic:

```
import { Component } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { SomePage } from '../some-page/some-page.page';

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor(private modalCtrl: ModalController){
```

```
  }

}
```

The first thing you will notice is the `import` statements. Anything that is required by the class that you are creating will need to be **imported**. In this case, we are importing **Component** from **@angular/core** which allows us to use the **@Component** decorator, and **ModalController** from the Ionic library which we can use to create modal overlays (i.e. pop up a page on top of our current page)

We are also importing **SomePage** which is a class/component of our own creation. The path for this simply follows the directory structure of your project, in this case, we have the **SomePage** component defined inside a folder called **pages** which is one level above the current file. The import should link to wherever the **.ts** file is for the class, but it is not necessary to include the **.ts** extension. If the file is in the same folder as the file you are coding in, you can reference it with `./the-file`. If you need to go up folders to link to the file you just use `../`, so if the file was up two folders, and then inside of a folder called `my-cool-page` you would do this: `../../my-cool-page/the-file`.

Next up we have the decorator, which we use to define the selector (i.e. the name this component will have in our DOM (Document Object Model), `<app-page-home></app-page-home>`) and the template.

Once we get past the decorator, we finally arrive at the class itself. Notice though that it is preceded by the **export** keyword, e.g:

```
export class HomePage {

}
```

The **export** keyword works in tandem with the **import** keyword, so we **export** classes that we want to **import** somewhere else. The last thing we have left to discuss is the constructor. We've already talked about the role of a constructor in classes in general, and it is no different here: the code inside of the constructor is run automatically when the class is instantiated, and it is where "setup" work is often executed.

There's a little bit more to it that you need to know, though. In Ionic we need to inject any services that we want to use inside of this class into the constructor, which looks like this:

```
  constructor(private modalCtrl: ModalController) {

  }
```

In this example, we want to use the **ModalController**, so we inject it into the constructor and then we can use it within the constructor.

In most cases, you will want to use the services you inject outside of the `constructor`. To make the service available to any function within the entire class, you must **set it as a member variable**. We *could* set up a member variable for the service in the `constructor` like this:

```
import { Component } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { SomePage } from '../some-page/some-page.page';

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  private modalCtrl: ModalController;

  constructor(modalCtrl: ModalController){
    this.modalCtrl = modalCtrl;
  }

  someOtherMethod(){
    this.modalCtrl.someMethodOfModalCtrl();
  }

}
```

We inject the service into the constructor, but we have also declared a member variable called `modalCtrl` and then we assign that injected reference to it with `this.modalCtrl = modalCtrl`.

This is unnecessary work, though. If we just prefix the dependency injection with the `private` or `public` keyword, it will automatically set up a member variable for us:

```
import { Component } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { SomePage } from '../some-page/some-page.page';

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  someOtherMemberVariable: string = "hey!";

  constructor(private modalCtrl: ModalController){

  }

  someOtherMethod(){
    this.modalCtrl.someMethodOfModalCtrl();
  }

}
```

Also note that any variables declared above the constructor like

`someOtherMemberVariable` is here will also automatically be set up as member variables. So we could access `someOtherMemberVariable` from anywhere in this class through `this.someOtherMemberVariable`. Any services you need to inject (and possibly set up as member variables) should be added to the constructor, any member variables you want to create for any other purpose should be declared above the constructor. If this concept sounds confusing to you right now it should make a little more sense when we start going through some examples.

Now we can access **modalCtrl** from anywhere in this class through `this.modalCtrl`. If we did not set up this member variable and tried to access **modalCtrl** from our `someOtherMethod` function, it would not work.

Creating a Page

Pages will usually make up a large portion of your application - for each screen you want to have in your application you will have a separate **Page** defined. Although we refer to them as "pages" they are just normal Angular components that use the **@Component** decorator.

As we've already discussed, the class for a page might look something like this:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
```

```
})
export class HomePage {

  constructor(){

  }

}
```

and the template file we are referencing in the decorator might look something like this:

```
<ion-header>
 <ion-toolbar>
  <ion-title>
    My Page
  </ion-title>
 </ion-toolbar>
</ion-header>

<ion-content>

  <ion-list>
    <ion-item>I</ion-item>
    <ion-item>Am</ion-item>
    <ion-item>A</ion-item>
    <ion-item>List</ion-item>
  </ion-list>
```

```
</ion-content>
```

The template file makes up what the user will actually see (we're going to discuss templates in a lot more detail later). The template file and the class work in unison: the class defines what template is to be shown to the user, and the template can make use of data and functions available in the class.

We've covered the basic structure of what a **Page** class looks like and what the `constructor` function does, but you can also add other functions that your page can make use of, for example:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage implements OnInit {

  constructor() {
    //this runs immediately
  }

  ngOnInit() {
    //this runs once the component has been initialised
```

```
  }

  someMethod(){
    //this only runs when called
  }

  someOtherMethod(){
    //this only runs when called
  }

}
```

You could call these other methods from your own code, or you could even have them triggered by a user clicking a button in the template, for example. These additional functions can be added to any class, it's not just specific to pages. We will cover these concepts in much more detail later, for now, we just want to understand the basic structure of the different class types and what they do.

NOTE: It is considered bad practice to do too much "work" in the `constructor` function. The constructor should only be used for basic set up, if you want to do more complicated things, you should use something like the `ngOnInit` hook instead. This is one of Angular's component lifecycle events that get triggered automatically.

NOTE: You can auto-generate a page in your project using the command `ionic g page MyPage`

Creating a Component

The code for a generic component looks a lot like the code for a page (remember, a page is just a component), just as it will look like just about everything else we create. When we create "pages" we are generally referring to components that will act as a particular view/screen and take up the whole page. But we can also create components in a similar manner to implement smaller features like a custom date picker, an animated button, or a rating component.

There is really no difference between a "page" component and any other component, apart from how we intend to use it. If we were to create a custom component that we did not intend to use as a "page" in our application, it might look like this:

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

@Component({
  selector: 'my-component',
  templateUrl: 'my-component.component.html',
  styleUrls: ['my-component.component.scss']
})
export class MyComponent {

  private text: string = "Hello World";

  constructor() {

  }

  someMethod(){
```

```
  }

  someOtherMethod(){

  }

}
```

This is exactly the same as what our pages look like, we just refer to it as a "component" instead of a "page". If we want to add that anywhere in our application, then we just reference its selector:

```
<my-component></my-component>
```

Let's also talk a little about the template for the component. This isn't really any different to how the template for a page works. We're referencing a file called **my-component.component.html** which might contain something like this:

```
<div>
  {{text}}
</div>
```

Just like with a page, we can reference data that is stored in the class definition. With this template, all our component will do is render:

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

into the DOM. Which is pretty boring of course, but you can create some pretty interesting, and reusable stuff with this functionality. Keep in mind that in order to be able to use a component, you need to make sure to import and declare the component either in your root module file, or the module file for a specific page (if your application is set up that way). We're getting a bit ahead of ourselves here, so don't worry too much if you don't really get the purpose of the module files. In brief, module files declare all the dependencies required (components, services, etc.) for specific areas of your application (the root module file being responsible for declaring what is required for the entire application). Depending on the structure of your application, you may have individual module files for each page.

If you wanted to add the component to your root **app.module.ts** file, you would do this:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { MyComponent } from './components/my-component/my-
component.component';

@NgModule({
```

```
  declarations: [
    AppComponent,
    MyComponent
  ],
  entryComponents: [
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    IonicModule.forRoot()
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported the component, and added it to the **declarations**. Keep in mind that if you have module files for each individual page in your application (and you will if you are using lazy loading as I will talk through later), then you would have to import the component into the module file for the specific page you want to use it on.

Once you have added it to the module file, then you can just reference it in the page's template like this:

```
<my-component></my-component>
```

It's actually pretty unlikely that you will need to create your own custom components like this in the beginning, as Ionic already provides most of what you would need (lists, tabs, buttons, inputs and so on). If you need something that Ionic does not already provide though, then you'll have to look at creating your own component.

NOTE: You can auto-generate a Component in your project using the command `ionic g component components/MyComponent`

I've skimmed over the role of module files a little here, as we will be covering it in more depth later. It is not critical that you understand everything that I have just mentioned at this stage.

Creating a Directive

As I mentioned before, components and directives are very similar, but in general, a component is used to create a completely new element, and a directive is used to modify the behaviour of an existing one.

The class for a custom directive looks like this:

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

@Directive({
  selector: '[my-directive]'
})
export class MyDirective {
```

```
  constructor() {

  }

}
```

In this directive, we also have a **selector** like we did for the component - but it's slightly different. Rather than representing the name of a tag, this can be used as an attribute on an element. You will do this a lot in Ionic, for example:

```
<ion-button color="light">
```

but in this case, we're creating our own custom directive that we can use on anything, e.g:

```
<ion-button my-directive>
```

But notice we don't actually have a template for this directive. Although we usually refer to any feature in our applications as 'components', technically speaking a component consists of a class **and** a template (view) - if it does not have a view then it is not a component.

I wanted to just cover the basics here, but I think it's also useful to know about **ElementRef**. This will give you access to the element that the directive was added to. You

can include it in your directive like this:

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

@Directive({
  selector: '[my-directive]'
})
export class MyDirective {

  constructor(private element: ElementRef) {

  }
}
```

By having a reference to the element the directive is attached to, you can modify its appearance/behaviour.

You will also need to add directives to a module file in order to be able to use them, just like the custom component we talked about.

NOTE: You can auto-generate a Directive in your project using the command `ionic g directive MyDirective`

Creating a Pipe

Pipes might seem a little complex at first glance, but they're actually really easy to

implement. They look like this:

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

@Pipe({
  name: 'myPipe'
})
export class MyPipe implements PipeTransform {

  transform(value, args?) {

    //do something to 'value'

    return value;

  }

}
```

The idea is that whatever you are passing through the pipe will go to this **transform** function, you do whatever you need to do to the value, and then you **return** the new value back. This new value will be rendered out to the screen, rather than the initial value. You can use it in your templates like this:

```
<p>{{someValue | myPipe}}</p>
```

which will run whatever **someValue** is through your custom pipe before displaying it. Once again, make sure you import and declare the pipe in the appropriate module file before you use it.

NOTE: You can auto-generate a Pipe in your project using the command `ionic g pipe MyPipe`

Creating an Injectable/Service/Provider

An `@Injectable` allows you to create a service that you can use throughout your application (like an interface between your application and an external or internal data service). Injectables might also be referred to as 'Providers' or 'Services'. An `@Injectable` service in Ionic might look like this:

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

@Injectable({
  providedIn: 'root'
})
export class DataService {

  constructor(private storage: Storage) {

  }

  getData(): Promise<any> {
```

```
    return this.storage.get('myData');
  }

  save(data): void {
    this.storage.set('myData', data);
  }

}
```

This code creates a provider/service called **DataService**. In this case, we are using it to save data to storage, and also to provide us access to that saved data.

In order to use this service, we would just need to import it into whatever component we want to use it in, and then inject it through the `constructor`, i.e:

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

@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor(private dataService: DataService){
```

```
  }

}
```

Now we could make use of this service anywhere in our home page. If we wanted to grab the saved data the service provides through the `getData` function, we would just do this:

```
const data = await this.dataService.getData();
console.log(data);
```

Notice that we use `await` here because it is a promise that is returned, so we need to wait for the promise to complete before we can access the data. If we want to use the service to save some data, then we would just do this:

```
this.dataService.save(someData);
```

Of course, you can use a provider to do other things besides fetching data - but that's a very common use case.

NOTE: You can auto-generate an Injectable in your project using the command `ionic g service MyProvider`

Summary

We've taken a pretty broad and basic look at how to create the different types of classes in Ionic, and there's certainly a lot more to know. But you should know enough now that it won't all look weird and foreign to you when we start diving into some real examples. The main goal of this basics section is to get you acquainted with the basic concepts, you don't need to be too concerned about having a complete understanding at this stage.

Lesson 6: Templates

Templates, I think, are one of the most fun bits of building Ionic applications. It's where the power of using a library that provides powerful web components really shines, as we can pretty much just drop stuff into place with HTML syntax. Creating interactive elements like a scrolling list doesn't require complicated JavaScript, and creating beautiful elements like a nicely styled card element doesn't require complicated CSS - with web components, you can just drop what you need into place. Take this code for example:

```
<ion-header class="ion-no-border">
  <ion-toolbar color="secondary">
    <ion-title> My Friends </ion-title>

    <ion-buttons slot="end">
      <ion-button (click)="doSomethingCool()"
        ><ion-icon slot="icon-only" name="add-circle"></ion-icon
      ></ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-searchbar (input)="getItems($event)"></ion-searchbar>
  <ion-list lines="none">
    <ion-item *ngFor="let item of items">
```

```
      <ion-avatar slot="start">
        <img [src]="item.picture" />
      </ion-avatar>
      <ion-label>
        <h2>{{item.name}}</h2>
        <p>{{item.description}}</p>
      </ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

with no additional styling, the code above would look like this right out of the box:

My Friends
Search
75 x 75
Josh
hello
75 x 75
Josh
hello
75 x 75
Josh
hello

It doesn't look amazing, but we already have a pretty complex layout set up with just a few lines of code, throw a bit of custom styling in and we'd have a pretty sleek interface. We're going to go through different aspects of creating templates in Ionic more thoroughly in just a moment, but I wanted to give you a sense of what a full template for a page might look like, and also how easy it is to use the components provided by Ionic.

There's a lot more to know about template syntax in Ionic, and it's not always as easy as just dropping in one line of code, so we're going to dive into templates in a lot of detail.

The * Syntax

Perhaps one of the most confusing things about the template syntax in Ionic/Angular applications is this little guy: `*`. You'll often come across code that looks like this:

```
<ion-item *ngFor="let item of items"></ion-item>
```

or

```
<p *ngIf="someBoolean"></p>
<p></p>
```

and so on. In Angular, the `*` syntax is used for **structural directives** which are a shortcut

for creating an embedded template, so if we use `*ngIf` as an example, the above could be expanded to:

```
<template [ngIf]="someBoolean">
  <p></p>
</template>
```

The reason for this use of templates is that Angular treats templates as chunks of the DOM that can be dynamically manipulated. So, in the case of `*ngIf` we don't want to just literally render out:

```
<p *ngIf="someBoolean"></p>
```

to the DOM. We want Angular to apply some logic to this, and then we want to render out:

```
<p></p>
```

if `someBoolean` is true, and nothing if it is false. Similarly, if we were to use `*ngFor` we don't want to literally render out:

```
<p *ngFor="let item of items">{{item.name}}</p>
```

we want to render out paragraph tags stamped with the information for each particular item that we are looping over:

```
<p>Bananas</p>
<p>More Bananas</p>
<p>Pancakes</p>
```

We need to use `<template>` to allow for this functionality, but writing out templates and applying our own logic to them manually is a lot of work, and the `*` syntax makes this a lot easier. In short, a "structural directive" is a directive that affects the structure of the application.

Now that we've covered the basic concept of what a structural directive is, let's jump into some specifics of how to use structural directives like `*ngIf` and `*ngFor`.

Looping

Quite often, you will want to loop over a bunch of items - when you have a list of articles and you want to render all of the titles into a list for example. We can use the **ngFor** directive which is supplied by Angular to achieve this - it looks something like this:

```
<ion-list>
  <ion-item *ngFor="let article of articles"
(click)="viewArticle(article)">
    {{article.title}}
  </ion-item>
</ion-list>
```

In this example, we create an `<ion-list>` and then for every **article** we have in our **articles** array we add an `<ion-item>`. I mentioned before (in the basics section) the use of `let` to create a local variable and we are using that here. This allows us to access whatever **article** the loop is currently up to, and we are using that to grab the title of the current article and render it in the list, and also to pass it into the `viewArticle` function that is triggered when the item is clicked.

By passing a reference to the current article to the `viewArticle` function we would then be able to do something like trigger a new page with the specifics of that article on it.

The key thing to remember with `*ngFor` is that you supply it an array of data, and then the `*ngFor` will repeat the element that it is attached to for each item in that array of data.

Conditionals

Sometimes, you will want to display certain sections of the template only when certain conditions are met, and there are a few ways to do this.

```
<div *ngIf="someBoolean"></div>
```

`*ngIf` will render the node it is attached to only if the expression evaluates to be `true`. So in this case, if `someBoolean` is true, the `<div>` will be added to the DOM, if it is `false` then it will not be added to the DOM.

`*ngIf` is great for boolean - true or false - scenarios, but sometimes you will want to do multiple different things based on a value. In that case, you can use `*ngSwitch`:

```
<div [ngSwitch]="paragraphNumber">
  <p *ngSwitchCase="1">Paragraph 1</p>
  <p *ngSwitchCase="2">Paragraph 2</p>
  <p *ngSwitchCase="3">Paragraph 3</p>
  <p *ngSwitchDefault>Paragraph</p>
</div>
```

In this example, we are checking the value of `paragraphNumber` with `*ngSwitch`. Whichever `*ngSwitchCase` statement the value matches will be the DOM element that will be rendered, and if none match the `*ngSwitchDefault` element will be used.

Another method to display or hide certain elements based on a condition is to use the `hidden` property. For example:

```
<ion-avatar [hidden]="hideAvatar" slot="start"></ion-avatar>
```

In this example, if the `hideAvatar` expression evaluates to be **true** the element will be hidden, but if it is **false** then it will be rendered. Using this method, you would need to have a `hideAvatar` member variable in your class definition which you could toggle to hide and show these elements. This is called a property binding, by using the square brackets we are able to reference values from the class, without the square brackets it would treat "hideAvatar" literally.

This method of hiding elements is less preferable. When you hide an element with `*ngIf` it is completely removed from the DOM, but when you hide it using the `hidden` attribute, the element is only hidden with CSS (but the element will still exist). The more lightweight the DOM is in your application, the better, so you should always use `*ngIf` where possible.

As well as conditionally displaying an entire element, you could also attach different classes to an element based on a condition, for example:

```
<ion-avatar [class.my-class]="showMyClass" slot="start"></ion-
avatar>
```

This is a similar concept to the `[hidden]` method above, but instead of showing and hiding the element based on a condition, it will add a class you have defined in your CSS based on a condition. In this case, we are saying we want to apply the `my-class` class if `showMyClass` evaluates to be true. This can be very useful, for example, you might want

to use it to style items that have already been read by the user with a different colour.

Slots

You may have seen multiple references to a `slot` attribute being attached to elements. This is a way to control how particular elements are displayed. When the Ionic team creates their web components, they are able to use "slots" which are kind of like placeholders for where content will be injected.

If we take the `<ion-buttons>` component as an example:

```
<ion-buttons slot="start"> // buttons go here </ion-buttons>
```

This is used inside of the `<ion-toolbar>` component. When we add buttons to the toolbar, we may want them displayed on the left side of the toolbar, or on the right side of the toolbar (or both). So, when the toolbar component is designed, "slots" are added to both of these areas. We can then specify which "slot" we want to inject our buttons into. You will see this pattern of either `slot="start"` or `slot="end"` used on a lot of components to specify their position. However, "start" and "end" are not the only kinds of slots. For example, for an `<ion-icon>` we might specify `<ion-icon slot="icon-only">` to indicate that this icon is being used as a standalone button with no associated text.

It won't be immediately clear what "slots" are available, or what they do, so you will often need to refer to the documentation for the specific component to figure out how to style

your component correctly. Just go to the slots section of the documentation for the component you are working with.

Ionic Web Components

Everything we've covered above is general Angular and web component stuff, there is nothing specific to Ionic there (except for where we are using Ionic specific components as an example). Now we're going to go through some of the Ionic specific stuff, starting with the basic layout of an Ionic page template:

```
<ion-header>
  <ion-toolbar>
    <ion-title> Ionic Blank </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding"> </ion-content>
```

This is the automatically generated template you will get when generating pages in Ionic. All of the elements here are **web components** which are provided by Ionic. As we have discussed, a web component is basically a custom HTML element with styles and behaviours attached to it. This allows us to drop complex mobile user interface elements into our templates easily (since Ionic has already done all the hard work for us).

Remember that since they are just web components, you can use them *anywhere*, we just happen to be using them in an Angular application. When we create our own components

in Angular - all of the pages we create are components, for example - they are **Angular components** not **web components**. They behave in a very similar fashion, but the difference is that a web component can be used anywhere, and an Angular component can only be used in an Angular application (side note: it is also possible to deploy Angular components as web components using Angular Elements).

Let's talk through what some of these components are actually doing for us.

The `<ion-content>` element is a component you will see on just about every page, and it is used to hold the main content of the page. We can add whatever content we like here, and it will enable smooth scrolling for us.

The `<ion-toolbar>` is also common, and is what adds the header bar to the top of the page, where you can place the title of the page as well as buttons on the left or right of the navigation bar (using "slots" to place the buttons however you wish). You will also find that Ionic has page transition animations built-in by default, so when you navigate between pages the toolbar will animate as you would expect a normal native application to.

The `<ion-title>` is less impressive, but allows us to add a title to the page that will be styled appropriately. These are the basic components you will find on most pages, but there is *a lot* more that we can drop into our pages.

In the following sections, we are going to look at how we might use some of Ionic's other built-in components. We're not going to be covering anywhere near all of them because there is so many (we will cover more in the application walkthroughs later), I just want to give you a taste so you can start to get a feel for what is available. For a full list of all the available components, take a look at the Ionic documentation.

Lists

Lists are one of the most used components in mobile applications, and they provide an interesting challenge. That smooth scrolling you get when you swipe a list on native applications, with smooth acceleration and deceleration, that all just *feels right* - well that's *really* hard to replicate. Fortunately, you don't have to worry about it because Ionic does all the hard stuff for you, and using a list is as simple as adding the following to your template:

```
<ion-list>
  <ion-item><ion-label>Item 1</ion-label></ion-item>
  <ion-item><ion-label>Item 2</ion-label></ion-item>
  <ion-item><ion-label>Item 3</ion-label></ion-item>
</ion-list>
```

or if you wanted to dynamically create your list for a bunch of items defined in your class:

```
<ion-list>
  <ion-item *ngFor="let item of items"
(click)="itemSelected(item)">
    <ion-label>{{item.title}}</ion-label>
  </ion-item>
</ion-list>
```

Slides

Slides are another common element used in mobile applications, slides look like this:

where you have multiple different images or pages to show and the user can cycle through them by swiping left or right. Just like lists in Ionic, creating slides is really easy as well:

```
<ion-slides>
  <ion-slide>
    <h2>Slide 1</h2>
  </ion-slide>

  <ion-slide>
    <h2>Slide 2</h2>
  </ion-slide>

  <ion-slide>
    <h2>Slide 3</h2>
  </ion-slide>
</ion-slides>
```

A container of `<ion-slides>` is used, and then each individual slide is defined with `<ion-slide>`. It is also possible to supply some options to define the behaviour of the slides (I'll give you a more complete example later).

Input

In Ionic, rather than using `<input>` tags for user input you use the Ionic equivalent which is `<ion-input>`. Just like a normal `<input>` you can specify a type depending on what sort of data you are capturing, but by using the Ionic versions we will be taking advantage of the custom inputs Ionic has designed for mobile.

```
<ion-list>
  <ion-item>
    <ion-label>Username</ion-label>
    <ion-input type="text" value=""></ion-input>
  </ion-item>

  <ion-item>
    <ion-label>Password</ion-label>
    <ion-input type="password"></ion-input>
  </ion-item>
</ion-list>
```

As well as `<ion-input>` specifically, you will also find that Ionic provides other input elements like `<ion-select>`, `<ion-radio>`, `<ion-checkbox>` and `<ion-toggle>`.

Grid

The Grid is a very powerful component, and you can use it to create complex layouts. If you're familiar with CSS frameworks like Bootstrap, then it is a very similar concept. When placing components into your templates, in general things just display one after the other, but with the Grid, you can come up with just about any layout you can imagine.

It works by positioning elements on the page based on rows and columns. Rows display underneath each other, and columns (which are placed inside of rows) display side by side. For example:

```
<ion-row>
  <ion-col></ion-col>
  <ion-col></ion-col>
</ion-row>
<ion-row>
  <ion-col></ion-col>
  <ion-col></ion-col>
  <ion-col></ion-col>
</ion-row>
```

This will create a layout with two rows, the top row will have two columns and the bottom row will have three columns. By default, everything will be evenly spaced, but you can also specify how wide columns should be if you like:

```
<ion-row>
  <ion-col size="3"></ion-col>
  <ion-col size="1"></ion-col>
  <ion-col size="2"></ion-col>
  <ion-col size="6"></ion-col>
</ion-row>
```

This will create a single row with 5 different columns of varying widths based on a full page width of 12 columns. For more details take a look at the documentation.

Personally, I generally prefer to use standard CSS (like `flex`) to handle layouts within Ionic application (you could also use CSS Grid).

Icons

Icons are heavily used in most applications today, they are great because they allow you to communicate what something does using a metaphor rather than relying on text. It's good for usability (sometimes) and usually looks way better than using a button that says something like "Add Item" (but not always!).

Ionic provides a ton of icons that you can use out of the box, like:

```
<ion-icon name="heart"></ion-icon>
```

You just have to specify the name of the icon you want to use. They even have variations of the same icons so that they display differently between iOS and Android to better match the style of the platform. For a full list of available icons you can go here.

There is a ton more default components, and a lot more to know about even the ones I've mentioned here, so make sure you have a look through the documentation to familiarise yourself with what's available. We will, of course, also be covering more components, and in more depth, as we progress through the book.

Lesson 7: Styling and Theming

One thing I really like about Ionic is that the default components look great right out of the box. Everything is really neat, sleek and clean... but also maybe a little boring. I like simple, simple is great, but you probably don't want your app to look like every other app out there. Take the example we used in the templates lesson:

My Friends
Search
75 x 75
Josh
hello
75 x 75
Josh
hello
75 x 75
Josh
hello

We have a simple and clean interface, but it's probably not going to win any awards for its design. It uses the default styling with absolutely no customisation whatsoever. If you take a look at some of the example apps that I've included in this book you will see that they mostly all have custom styling:

animalsbeingderps
Not quite ready for bedtime
86
Useless hooman, belly ain't gona rub itself.

Some of the applications have simple styling, where just a few changes are made to achieve a much more attractive look. Some of the applications have more complex styling, that completely change the look and feel of the application.

In this lesson, I'm going to show you the different ways you can customise your Ionic applications and the theory behind theming in general.

Introduction to Theming in Ionic

When styling an Ionic application, there is nothing inherently different or special about it – it's no different than the way you would style a normal website. I often see questions like:

"Can I create [insert UI element/interface] in Ionic?"

and the answer is generally **yes**. Could you do it on a normal webpage? If you can then you can do it in Ionic as well.

A lot of people may be used to just editing CSS files to change styles, but there is some added complexity with Ionic/Angular, which is primarily due to the usage of **SASS**, **CSS Variables/Custom Properties**, and **Shadow DOM**. None of these technologies are specific to Ionic or mobile web app development – these concepts can also be used on any normal website – but many people may not be as familiar with these concepts as they are with basic CSS.

If you're not already familiar, **.scss** is the file type for **SASS** or **Syntactically Awesome Style Sheets**. If this is new to you, you should read more about what SASS is and what it does here. For those of you short on time, what you put in your **.scss** files is exactly the

same as what you would put in **.css** files, you can just do a bunch of extra cool stuff as well like define variables that can be reused in multiple areas. These **.scss** files are then compiled into normal **.css** files (it's basically the same concept we use in Ionic, where we code using all the fancy new ES6 features, but that is then transpiled into ES5 which is actually supported by browsers now).

Since Ionic 4, Ionic has switched to using CSS variables instead of SASS variables for controlling the applications theme, which was previously the primary feature SASS was used for. This means that if you don't really care to learn anything about SASS (like how to use loops), then you can more or less just pretend the **.scss** files are normal **.css** files.

One of the primary benefits of CSS variables is that they are native to the browser, and don't require precompiling like SASS variables do. A CSS variable is defined in CSS like this:

```
--my-background-color: #fff;
```

and then this variable can be reused wherever you like:

```
.some-class {
  background-color: var(--my-background-color);
}
```

You probably won't often create these variables yourself, but rather overwrite the default

Ionic CSS variables that look like this:

```
--ion-some-variable: 20px;
```

Ionic heavily uses CSS variables to define styles for their components, and by overwriting these variables we can easily alter what these components look like. We could just change just one variable and automatically update the styles across many different components.

When theming your application, you're mainly going to be editing your **.html** templates and **.scss** stylesheets – you will **NEVER** edit any .css files directly. The `.css` files are generated from the `.scss` files, so if you make any changes to the `.css` file it's just going to get overwritten. You can place any normal CSS you like in the `.scss` files, as well as any SASS syntax you like.

If you take a look at the files generated when you create a new Ionic project, you will see some global **.scss** files inside of your application, so let's quickly run through what their purpose is.

- **theme/variables.scss** - This file contains some of the default CSS variables that Ionic uses. You can modify things like the `primary` or `secondary` colours that your application uses. By changing a single variable in this file, you can change the styling of every Ionic component in your application that is using that variable. This means you could easily switch the entire theme of your application just by changing a few variables. Although there are some default variables displayed in this file for theming, you can use this file to globally change *any* Ionic CSS variable - just add it to the file

and set whatever value you like.

- **global.scss** - This file is simply used to supply any styles that are being used globally throughout your application. You might want to define some CSS classes in here that you reuse throughout multiple places in your application.

On top of these **.scss** files, you will also have a `.scss` file for each component you create (or at least you should). To refresh your memory, the components you create in Ionic will contain these three key files:

my-component

- my-component.page.ts
- my-component.page.html
- my-component.page.scss

We have the class definition in the `.ts` file, the template in the `.html` file and any styles for the component in the `.scss` file. Although it's not strictly required, you should always supply the styling for a component inside of its own `.scss` file rather than the `global.scss` file.

Why? Well you *could* just put all of your styles in the **global.scss** file and everything would work exactly the same, but there are two major benefits to splitting your styles up in the way I described above:

Organisation – Splitting your code up in this way will keep the size of your files down, making it a lot easier to maintain. Since all of the styles for a particular component can be found in that components `.scss` file, you'll never have to search around much.

Modularity – one of the main reasons for the move to this component-style architecture in Angular and Ionic is modularity. Before, code would be very intertwined and hard to separate and reuse. Now, almost all the code required for a particular feature is contained within its own folder, and it could easily be reused in other projects.

Now that we've gone over the theory, let's look at how to actually start styling our Ionic applications.

Methods for Theming an Ionic Application

I'm going to cover a few different ways you can alter the styles in your application. It may seem a little unclear what way to do things because in a lot of cases you could achieve the same thing multiple different ways. In general, you should try to achieve what you want to do without creating custom styles (which we will cover last here). Instead, you should first try using the pre-defined attributes or overriding SASS variables. If it can not be done any other way, then look into creating your own custom styles. Don't worry too much though, just try to keep things as simple as you can.

1. Attributes

One of the easiest ways to change the style of your application is to simply add an attribute to the element you're using. As I mentioned above, CSS variables are used to define some theme colours, and these are:

- primary (--ion-color-primary)
- secondary (--ion-color-secondary)
- tertiary (--ion-color-tertiary)

- success (--ion-color-success)
- warning (--ion-color-warning)
- danger (--ion-color-danger)
- light (--ion-color-light)
- medium (--ion-color-medium)
- dark (--ion-color-dark)

which you can see defined in the **theme/variables.scss** file:

```
/** Ionic CSS Variables **/
:root {

  /** primary **/
  --ion-color-primary: #e74c3c;
  --ion-color-primary-rgb: 72,138,255;
  --ion-color-primary-contrast: #fff;
  --ion-color-primary-contrast-rgb: 255,255,255;
  --ion-color-primary-shade: #3f79e0;
  --ion-color-primary-tint: #5a96ff;

  /** secondary **/
  --ion-color-secondary: #32db64;
  --ion-color-secondary-rgb: 50,219,100;
  --ion-color-secondary-contrast: #fff;
  --ion-color-secondary-contrast-rgb: 255,255,255;
  --ion-color-secondary-shade: #2cc158;
  --ion-color-secondary-tint: #47df74;
```

```
  /* etc. etc. */

}
```

As you can see above, Ionic provides some defaults for what these colours are, but you can also override each of these to be any colours you want. Since these are defined inside of the `:root` pseudo-selector, the variables will apply globally to your application. However, you could also define different values for these variables inside of other selectors so that any elements that match that selector would be given a different value, e.g:

```
my-red-page {
  --ion-color-primary: red;
}
```

With these variables defined, if you add the **primary** attribute to particular elements they will get the `--ion-color-primary` colour applied, or if you add the **danger** attribute it will get the `--ion-color-danger` colour applied.

To give you an example, if I wanted to use the **secondary** colour on a button I could do this:

```
<ion-button color="secondary"></ion-button>
```

or if I wanted to use the secondary colour on the toolbar I could do this:

```
<ion-toolbar color="secondary"></ion-toolbar>
```

However, this will **only work on elements that support the color property**. You can view the documentation for a particular component to see what properties it supports.

Keep in mind that properties are not limited to just changing the colour of elements, some properties will affect different styles. For example, if we wanted to remove the borders from a list component we could do this:

```
<ion-list lines="none"></ion-list>
```

or we could even determine whether a list item should display an arrow to indicate that it can be tapped:

```
<ion-item detail></ion-item> <ion-item detail="false"></ion-item>
```

There are a bunch more of these properties, so make sure to poke around the documentation when you are using Ionic's built-in components. Just specifying `lines="none"` is a really easy way to remove lines from a list, but if you didn't know this attribute existed (which is quite possible) then you'd likely end up creating your own

custom styles unnecessarily. This is why I recommend trying to do things with attributes first if you can because you could save yourself a lot of effort. However, it's not really that big of a deal if you don't know about a particular attribute and just add the style yourself (as we will discuss later)

2. CSS Variables

The next method you can use to control the style of your application is to change the default CSS variables (like editing the `--ion-color-whatever` variables we talked about above). These are really handy because it allows you to make app-wide style changes to specific things. I touched on this before, but the basic idea with CSS variables is that you can define a variable like this:

```
--my-color-variable: red;
```

and then you can use this elsewhere in your application. If you wanted to make the background colour on 20 different elements red, rather than doing:

```
background-color: red;
```

for all of them, you could instead do this:

```
background-color: var(--my-color-variable);
```

The benefit of this is that now if you wanted to change the background color from red to green, all you have to do is edit that one variable – not every single class you have created. This is why you'll find that variables are named in the manner of **primary** and **danger** rather than specifically **blue** and **red**. There may come a time when you want to change your primary colour to be purple, but if you give variables specific names like `--my-blue-color` and you change it to be purple it's going to make your code pretty confusing.

You probably won't be creating many of your own variables, but Ionic defines and uses a bunch of these variables, and you can easily overwrite them to be something else. Let's take a look at a few:

- **--ion-color-primary**
- **--ion-background-color**
- **--ion-toolbar-background-color**
- **--ion-item-background-color**

You can look at the documentation for more information on these and what they default to, but the names of the variables do make it pretty clear. Besides looking at the documentation for each component (under the CSS Custom Properties section), a good way to find variables that affect the components you are working on is just to `Right Click > Inspect Element` and look at the styles that are being applied to the element. If a CSS variable is being used, you will see it listed directly in the debugging window, e.g:

```
background-color: var(--ion-item-background-color);
```

and you can even edit the value for this variable directly through the debugger if you want to test your changes live.

Editing these variables in your application is really simple, just open the **theme/variables.scss** file and add your own variable definitions inside of the `:root` pseudo-selector. For example:

```
:root {
  --ion-item-background-color: #1f1e1e;
  --ion-background-color: #1f1e1e;

  /* Default Ionic color variables go here */
}
```

If you don't want the variable change to apply to the entire application, then you can also supply the variables to other selectors instead. If you just wanted to apply a different background colour to the home page, then you could do something like this in your **home.page.scss** file:

```
ion-content {
  --ion-background-color: #000;
}
```

The great thing about editing these default CSS variables is that you can, with one change, make all the changes necessary everywhere in the app. Some variables use the values of other variables, so if you wanted to just do this manually with CSS you would probably need to make a lot of edits to get the effect you wanted (not to mention that a lot of Ionic's components *require* that you use CSS variables to style them, more on that in a moment).

3. Custom Styles

We've talked about using CSS variables and applying properties to change the style of elements. Given that you can override these variables to whatever you like, it's a good approach to set the `primary`, `secondary`, `danger` etc. variables to match the colour palette of your design, and then use those to set the styles of elements, rather than defining custom CSS classes.

But, sometimes there will come a time where you need to define some plain old CSS classes to achieve what you want. Or maybe you just want to do everything this way because that's what you are more comfortable with. You can either define these custom classes in the **global.scss** file if the class will be used throughout the application, or in an individual components `.scss` file if it is only going to be used for one component.

You can also define custom styles on the element directly by using the `style` tag, but make sure you use this sparingly.

As you can see, there are a few different ways you can change the styling of your Ionic applications. In general, it's best to do as little as possible to achieve what you need. Try to

achieve as much as you can with properties and CSS variables because it will make your life easier (especially if you want to change things down the road).

An important thing to keep in mind is that Ionic seamlessly adapts the user interface design to match the conventional styling of the platform it is running on, so the more "hacky" or "brute force" your solution for styling is, the greater chance you have of breaking this behaviour (or at least making things more difficult for yourself).

Keep the Shadow DOM in Mind

Some of Ionic's web components make use of something called "Shadow DOM". This is a reasonably complex concept, and it isn't really something you need to understand fully if you don't want to. If you are interested in knowing more about it, you might want to read this article on my blog: [Shadow DOM Usage in Ionic Web Components](#).

I just want to provide you with a very basic overview here. The idea of a Shadow DOM is to provide Ionic's web components an isolated environment to work in - the web components styles won't affect the rest of your page, and your pages styles won't affect Ionic's web components. This is generally a good thing, but it can cause some issues if you *want* to affect the styles of Ionic's web components.

When you use a web component like `<ion-item></ion-item>` it injects more than just those tags into the DOM. In fact, when you run your application it looks more like this if you were to open up your debugger:

```
<ion-item>
  <style></style>
```

```
  <div class="item-native">
    <slot></slot>
    <div class="item-inner">
      <div class="input-wrapper"></div>
      <slot></slot>
    </div>
  </div>
</ion-item>
```

In previous versions of Ionic (that didn't use Shadow DOM) if we wanted to we could set up a custom style to target the internals of the component, e.g:

```
ion-item .item-inner {
  padding: 20px;
}
```

However, this no longer works because everything inside of `<ion-item>` is protected from interference by a Shadow DOM. If you do want to make a style change to the internal elements of one of Ionic's web components that use Shadow DOM (not all of them do) then you have to overwrite the CSS variables that Ionic is using, e.g:

```
ion-item {
  --ion-item-background-color-active: #000;
  --min-height: 100px;
```

```
}
```

You can target the web component itself with standard CSS properties (e.g. `ion-item`), but you can not target anything *inside* of the web component. To see what CSS variables are available for styling the internals of Ionic components, you can either find them through inspecting the elements Shadow DOM with Chrome DevTools, or you can view the list of available CSS Custom Properties in the documentation. The `<ion-item>` component, for example, allows the following CSS variables to be set:

```
--background
--background-activated
--background-activated-opacity
--background-focused
--background-focused-opacity
--background-hover
--background-hover-opacity
--border-color
--border-radius
--border-style
--border-width
--color
--color-activated
--color-focused
--color-hover
--detail-icon-color
--detail-icon-font-size
```

```
--detail-icon-opacity
--highlight-color-focused
--highlight-color-invalid
--highlight-color-valid
--highlight-height
--inner-border-width
--inner-box-shadow
--inner-padding-bottom
--inner-padding-end
--inner-padding-start
--inner-padding-top
--min-height
--padding-bottom
--padding-end
--padding-start
--padding-top
--ripple-color
--transition
```

Shadow Parts

Shadow Parts are a newer way to help style elements that are within a Shadow DOM. Because this is a newer API, you might first want to check that Shadow Parts are supported by the browsers you intend to target: browser support (both Chrome for Android and iOS Safari support the API).

To reiterate, the basic idea of using a Shadow DOM in the context of styling is to ensure

that the component won't be effected by styles outside of the component. It might be important that a particular component has a `padding` of `5px` on a particular element, and we wouldn't want a global style that sets the padding to `20px` to interfere with that.

Shadow Parts are basically a way for the designer of the component to mark particular elements within that component as being safe to target with CSS rules, and to expose them to styles coming from outside of the Shadow DOM.

Let's take the `<ion-content>` component as an example. If we look at the internal structure of the `<ion-content>` component (you can do this just by opening it's Shadow Root in the Chrome DevTools Element inspector) we would see something like this:

```
<ion-content>
  <!-- shadow-root -->
    <div id="background-content" part="background"></div>
    <main class="inner-scroll scroll-y overscroll" part="scroll">
    <div class="transition-effect"></div>
    <slot name="fixed"></slot>
  <!-- shadow-root -->
</ion-content>
```

All of these elements, except the `<ion-content>` itself, are within the Shadow DOM. That means we can't target them with standard CSS properties. If we were to try to set the background colour of `background-content` like this:

```
ion-content #background-content {
  background-color: #000;
}
```

...it wouldn't work. However, notice that a couple of these elements have a `part` attribute supplied? This is because they are using the Shadow Parts API. Anything that is exposed as a "part" can actually be styled directly, we just have to target it with a special selector syntax:

```
ion-content::part(background) {
  background-color: #000;
}
```

All we need to do is target the component itself, and then supply the `part` name to `::part()`. Then we can just style using normal CSS properties. Just like with the CSS Variables, a list of available parts is also supplied in [the documentation for each component](#).

Lesson 8: Navigation, Routing, and Lazy Loading

We're going to be getting into something a little more technical now. In this lesson, we will be covering the concepts that will allow you to navigate between (and pass data between) different pages in your application. Once you've got the concept down, performing navigation in your application will be quite easy, but there is a bit of upfront configuration involved that may look a little confusing.

I want to reiterate that this "Basics" section of the book is very theory oriented. The main goal is to introduce you to these concepts, but I don't expect you to fully comprehend them right away. We will be putting these concepts into practice later in the book. It's also worth noting that most of the tricky set up work will already be set up for you in the starter templates, so all you will need to do is make your own modifications to it. However, we are still going to cover the concepts in-depth here so that you are able to have an understanding of what is actually happening to make your application work.

The particular style of navigation that we will be covering is something that was introduced in the Ionic 4.x update. Some of you may be reading this from the perspective of having implemented Ionic navigation in past versions of Ionic, and some of you will be completely new to Ionic. I will be making comparisons to previous versions of Ionic for the benefit of people who are already familiar with the "old" style of navigation in Ionic, but if you are completely new to Ionic then you can just ignore those references.

Angular Routing

Before we get into various routing concepts, keep in mind that although Angular style

routing will be the default for applications, the typical push/pop style navigation that Ionic has used in the past **will still be available**, and you can even use this style of navigation directly through Ionic's `<ion-nav>` web component. However, the recommended approach moving forward will be to use the Angular Router for Ionic/Angular applications. This is what we will be using for all of the examples in this book.

The Ionic team is aiming to make Ionic more generic so that it isn't tied to any specific framework, and implementing their own navigation/routing for each framework (as they have done for Angular in the past) would get quite messy and would ultimately be somewhat unnecessary. Instead, you should just rely on the native navigation for whatever framework it is that you are using Ionic with.

Keep in mind that although the way Angular routing will work with Ionic is mostly the same as standard Angular routing (and thus, you could apply any examples/advice for an Angular application to an Ionic/Angular application), Ionic does have its own router outlet implementation called `<ion-router-outlet>` (basically, you just plop the router outlet wherever you want the component for the active route to be displayed). This is mostly the same as Angular's `<router-outlet>` except that it will automatically apply the screen transition animations you would expect from a mobile application (just like in previous versions of Ionic).

The Router Outlet

The router outlet is a simple but important concept, so let's touch on that first. There is generally only ever one page shown on the screen at a time, and so we need a way to control what page it is that is being displayed to the user - this is what we use the `<ion-router-outlet>` for.

If you were to look at **src/app/app.component.html**, which is the template for the **root component** of your application (i.e. the component that contains the entirety of your application) you will find this

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

We have just two components here. The `<ion-app>` is simply a container component for our entire Ionic application, and then the `<ion-router-outlet>` is used to display whatever page we want to be displayed at the time. The concepts we are about to discuss in the rest of this lesson will be used to control what this `<ion-router-outlet>` displays, and when it displays it.

Although we would never write the following code ourselves, if you inspect your application using the browser debugger when it is running, you will see how pages are injected into this outlet:

```
<ion-app>
    <ion-router-outlet>
        <app-page-home>
            <ion-header>etc.</ion-header>
            <ion-content>etc.</ion-content>
        </app-page-home>
```

```
    </ion-router-outlet>
</ion-app>
```

In this case, the application has determined that the `app-page-home` component is the component for the currently active route, and so that is what is displayed here.

Angular Routes

If you have been building Ionic applications with previous versions of Ionic, then you would be used to navigating through your applications by **pushing** new pages to navigate forward in the navigation stack:

```
this.navCtrl.push('SignupPage');
```

Popping pages to move backward in the navigation stack:

```
this.navCtrl.pop();
```

or starting a completely new navigation stack by changing the root page:

```
this.navCtrl.setRoot('HomePage')
```

If you are completely new to Ionic, then **the information above is of no relevance to you**, as we will be using Angular routing instead. Angular routing is different in that it is based on the browser model of navigation and uses the URL to determine which page/component to show. In your application, you would need to supply a set of routes that might look something like this:

```
const routes: Routes = [
  { path: 'login', component: LoginPage },
  { path: 'home', component: HomePage },
  { path: 'detail/:id', component: DetailPage },
  { path: '', redirectTo: '/login', pathMatch: 'full'}
];
```

The `path` property defines the URL, and the `component` property defines which component should be displayed by the `<ion-router-outlet>` when that URL is hit. If I were to go to the following URL:

```
http://localhost:8100/home
```

then the **HomePage** would be displayed. We also have a default route defined at the bottom so that if no route is supplied, it will use the `login` route.

NOTE: This is a simple approach to routing, make sure to also read the section on lazy

loading later in this lesson for a more "best practice" approach.

Navigation in the application is then based on whatever the URL currently is, and what route it matches. Simply changing the URL would change the current page, but in order to navigate in your application, you have a few options.

The simplest is to use a **routerLink** in the template to link to another page:

```
<ion-item routerLink="detail/{{item.id}}">
```

In this case, we would be launching the **DetailPage** and supplying it with an `id` route parameter (we could then use that data inside of the component). One our `{{item.id}}` interpolation is evaluated, the resulting link would be something like this:

```
/detail/12
```

Although only the `routerLink` is required to navigate, you can also add an additional `routerDirection` value of `root`, `forward`, or `back`:

```
<ion-item routerLink="detail/{{item.id}}"
routerDirection='forward'>
```

The problem with just changing routes in the URL is that it doesn't have a sense of direction, and when Ionic is animating your page transitions it needs to know the "direction" so that it can apply the animation properly. This just provides a way for us to let Ionic know what direction the navigation is - e.g. did we click to view more details about an item (that would be a forward navigation)? Or are we exiting out of a page to go back to the previous page (that would be a back navigation)?

You can also navigate programmatically by injecting the Angular `Router` into your pages and calling either of these methods:

```
this.router.navigateByUrl('/login');
```

or

```
this.router.navigate(['/detail', { id: itemId }]);
```

But again, the default Angular router doesn't allow us to specify the "direction" of the navigation that Ionic needs to know. It is generally better in an Ionic application to use `NavController` to perform these navigations instead - similarly, you can use the methods `navigateRoot`, `navigateForward`, or `navigateBack` to perform the navigation (you just supply the route as you normally would):

```
this.navCtrl.navigateRoot('/login')
```

```
this.navCtrl.navigateForward('/cars')
```

```
this.navCtrl.navigateBack('/categories')
```

Finally, for simple back navigation you will generally just have the default `<ion-back-button>` component in the header for your page:

```
<ion-buttons slot="start">
  <ion-back-button defaultHref="/home"></ion-back-button>
</ion-buttons>
```

This will automatically navigate back through the navigation history. However, since a user could go to any URL in your application directly (e.g. if you host the application as a PWA), we can get into a state where the user is on a page with a back button, but there is no history (because it is the first page they loaded). We provide a `defaultHref` for this situation so that in that particular scenario the back button would just default to going back to the `/home` route.

Lazy Loading

Aside from the potential future-proofing aspect of using Angular routing instead of the old push/pop style, there are a couple of big reasons to switch to it as soon as you upgrade to any version after Ionic 4.x:

- The `@IonicPage` decorator has been removed and it is no longer possible to enable lazy loading without Angular routing
- If you are developing a PWA, the browser-based navigation approach is much simpler

Lazy loading is the big ticket item here. If you are unfamiliar with lazy loading the basic idea is that it breaks your application down into smaller chunks, and only loads what is necessary at the time. When your application is first loaded, only the components that are necessary to display the first screen need to be loaded.

With lazy loading, your application can boot much faster because it doesn't need to load much - you could have a huge application, with 50 pages, but it could still load just as fast as an application with just 2 pages. Without lazy loading, you need to load the entire application upfront before anything can be done.

With smaller applications, it is not that much of a big deal, but it is critical for larger applications. I would generally encourage everybody to use lazy loading by default.

Lazy Loading with Angular Routing in Ionic

Lazy loading with Angular routes is not too dissimilar to regular routing. However, your routes would look like this instead:

```
const routes: Routes = [
  { path: 'login', loadChildren: () =>
import("./login/login.module").then((m) => m.LoginPageModule),
  { path: 'home', loadChildren: () =>
import("./home/home.module").then((m) => m.HomePageModule),
  { path: 'detail/:id', loadChildren: () =>
import("./detail/detail.module").then((m) => m.DetailPageModule),
  { path: '', redirectTo: '/login', pathMatch: 'full' },
];
```

Instead of supplying a `component` for the route, we supply a `loadChildren` property. Since we are not loading everything up front, this property is essentially saying "go to this file to determine what components need to be loaded". We link to the module file for the page (a module is basically just a way to tell Angular what needs to be loaded to make this feature work).

If we use the `home` route as an example, the `loadChildren` property is:

```
() => import("./home/home.module").then((m) => m.HomePageModule)
```

The router knows that it needs to open the **home.module.ts** file, and looks for the **HomePageModule**. That file might look like this:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { IonicModule } from '@ionic/angular';
import { FormsModule } from '@angular/forms';
import { HomePage } from './home.page';

import { HomePageRoutingModule } from './home-routing.module';


@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    HomePageRoutingModule
  ],
  declarations: [HomePage]
})
export class HomePageModule {}
```

NOTE: If you are using lazy loading, you should make sure that you are not also declaring these pages in your root module file (**app.module.ts**).

This specifies all of the dependencies required for this particular route, and then it also supplies its own `HomePageRoutingModule` to define any additional routes. That file might look like this:

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

const routes: Routes = [
  {
    path: '',
    component: HomePage,
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HomePageRoutingModule {}
```

In this case, we just want the **HomePage** loaded when we hit the `/home` route, so we supply the default route with an empty path (and this route will load the **HomePage** component). If you wanted to, you could also add additional routes in the **HomePageRoutingModule**. If you were to define a route with a path of `something` in this file, the user could then activate that route by going to `/home/something`, and you could use that to load a different component.

With this setup, we can make it so that Angular only loads the directives/components that we require for a particular route, rather than all of the directives/components for our entire

application. The set up might seem a bit convoluted, but like with the old style of lazy loading in Ionic 3, it's all pretty much the same for every page, and it will all likely be set up by default, so you don't really need to think too much about it - mostly, you will just need to make sure you are importing any additional modules your component needs to use in the `imports` here (e.g. the `FormsModule` or the `ReactiveFormsModule` if you are using forms).

Again, if this is all seeming a little overwhelming, don't worry. I'm giving you a lot of theory, and this stuff probably won't start to make sense until you've built at least a few applications. You are going to attempt to use a particular feature, like setting up some inputs for forms, and you are going to get an error that breaks your application. You will Google the error and figure "oh, I need to add FormsModule as an import to my home pages module file". With time, you will start to remember this stuff, I just don't want you to feel discouraged if you're not "getting it".

We will be covering more navigation and routing examples throughout the rest of this book.

Lesson 9: User Input

Not all mobile applications require user input, but many do. At some point, you're going to want to collect some data from your users. That might be some text for a status update, their name and shipping address, a search term, a title for their todo list item or anything else.

Whatever the data is, the user is going to be entering it into one of the templates in your application. To give you an example, in Ionic we could create a form in our template with the following code:

```
<ion-list>
  <ion-item>
    <ion-label>Username</ion-label>
    <ion-input type="text"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label>Password</ion-label>
    <ion-input type="password"></ion-input>
  </ion-item>

  <ion-button expand="full" (click)="login()">Sign in!</ion-
button>
</ion-list>
```

which would produce a simple login form that looks like this:

This would allow the user to enter some information into these input fields. However, we need to know how to get the data that is being entered into our `.html` template and make use of it in our `.ts` class. In this lesson, we are going to discuss a couple of different ways you can do that.

Two Way Data Binding

Two way data binding essentially links a value of an input field in the template, to the value of a member variable in the class. Take the following example:

Template:

```
<ion-input type="text" [(ngModel)]="myValue"></ion-input>
```

Class:

```
public myValue: string;

constructor(){

}
```

If we changed the value of the input in the template, then the `this.myValue` variable in the class will be updated to reflect that. If we change the value of `this.myValue` in the class, then the input in the template will be updated to reflect that. By using `ngModel` the two values are tied together: if one changes, the other changes.

Let's say you also had a submit button in your template:

```
<ion-input type="text" [(ngModel)]="myValue"></ion-input>

<ion-button (click)="logValue()">Log myValue!</ion-button>
```

When the user clicks the button we want to log the value they entered in the input to the

console. Since the button calls the `logValue` method when it is clicked, we could add that to our class:

```
logValue(){
  console.log(this.myValue);
}
```

This method will grab whatever the current value of the input is and log it out to the console. Rather than logging it out to the screen, you could also do something useful with it. This can be a convenient way to handle input because we don't need to worry about passing the values through a method, we can just grab the current values whenever we need.

It becomes a bit cumbersome when we have a lot of inputs, though, so it's not always the perfect solution. When dealing with more complex forms, we also have another option, which we will discuss now.

NOTE: When using `ngModel` you will need to make sure that you import the `FormsModule` from `@angular/forms` into the module file for your page.

Form Builder

Form Builder is a service provided by Angular, which makes handling forms a lot easier. There's quite a lot Form Builder can do but at its simplest it allows you to manage multiple input fields at once and also provides an easy way to validate user input (i.e. to check if

they actually did enter a valid email address).

To use Form Builder it needs to be imported (along with `FormGroup`) and injected into your constructor, e.g:

```
import { Component } from "@angular/core";
import { FormBuilder, FormGroup, Validators } from
"@angular/forms";

@Component({
  templateUrl: "my-details.html",
})
export class MyDetailsPage {
  private myForm: FormGroup;

  constructor(public formBuilder: FormBuilder) {}
}
```

Notice that **Validators** are also being imported here, which are what allow you to validate user input with Form Builder. Let's cover a really quick example of how you can use Form Builder to build a form. The most important difference with this method is that your inputs will have to be surrounded by a `<form>` tag with the `formGroup` property defined:

```
<form [formGroup]="myForm">
  <ion-item>
```

```
    <ion-label stacked>Field 1</ion-label>
    <ion-input formControlName="field1" type="text"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label stacked>Field 2</ion-label>
    <ion-input formControlName="field2" type="text"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label stacked>Field 3</ion-label>
    <ion-input formControlName="field3" type="text"></ion-input>
  </ion-item>

  <ion-button (click)="saveForm()">Save Form</ion-button>
</form>
```

You will also notice in the example above that we have defined the `formControlName` attribute on all of our inputs, this is how we will identify them with Form Builder in just a moment. Of course, we need a way to submit the form, so we've added a submit button and have also added a `(click)` listener on the form which calls the `saveForm` function. We will define that function in a moment, but for any of this to work, we first need to initialise the form in the constructor function for the page. This will look something like this:

```
constructor(public formBuilder: FormBuilder) {
  this.myForm = formBuilder.group({
```

```
    field1: [''],
    field2: [''],
    field3: ['']
  });
}
```

We simply supply all of the fields that are in the form (using the `formControlName` names we gave them) and provide the fields with an initial value (which we have left blank in this case). You can also supply a second value to each field to define a Validator if you like, e.g:

```
this.myForm = formBuilder.group({
  field1: ["", Validators.required],
  field2: ["", Validators.required],
  field3: [""],
});
```

Also, note that the variable `this.myForm` has to be the same as the value we supply for `[formGroup]` in the template. Now let's look at that `saveForm` method.

```
saveForm(){
  console.log(this.myForm.value);
}
```

To grab the details that the user entered into the form we can simply use `this.myForm.value` which will contain all the values that the user entered. If you have the complete version of this book, we will be walking through implementing some more advanced forms in the **CamperMate** application. Otherwise, if you'd like some more examples of using form inputs and validators, you can take a look at my [Advanced Forms & Validation in Ionic]() blog post.

Setting up forms using Form Builder is a little more complex, but it's much more powerful and worth the effort for more complex forms. For more simple requirements, using `[(ngModel)]` is fine in most cases.

NOTE: When using `FormBuilder` you will need to make sure that you import the `ReactiveFormsModule` from `@angular/forms` into the module file for your page.

Lesson 10: Saving Data

Let's say you've created an Ionic application where users can create shopping lists. A user downloads your application, spends 5 minutes adding their list and then closes the application... and all their data is gone.

Quite often when creating mobile applications you will want to store data that the user can retrieve later. A lot of the time this is done by storing the data somewhere that can be accessed through a remote server (think Facebook, Twitter, etc.) which would require an Internet connection and fetching the data remotely (which we will discuss in the next lesson). In many cases though, we may also want to store some data locally on the device itself.

There are a few reasons we might want to store some data locally:

- The application is completely self-contained and there is no interaction with other users, so all data can be stored locally
- We need to store data locally for some specific functionality like remembering logged in users
- We could skip unnecessary calls to a server by storing preference settings locally
- We could skip unnecessary calls to a server by caching other data locally
- We want to sync online and offline data so that the user can continue using the application even when they are offline (Evernote is a good example of this)

Ionic applications run in a browser, so we don't have access to the storage options that native applications do. We do still have access to the usual browser storage options that

websites have access to though like **Local Storage**, **Web SQL** (deprecated) and **IndexedDB**. These options might not always be ideal (for reasons I will cover shortly), but we can also access native data storage by using Capacitor which can overcome the shortfalls of browser-based data storage.

There are plenty of different options out there for storing data locally, but we are going to focus on the main ones when it comes to Ionic applications.

Local Storage

This is the most basic storage option available, which allows you to store up to **5MB** worth of data in the user's browser. Remember, Ionic applications technically run inside of an embedded browser even when they are built as iOS or Android applications.

Local storage gets a bit of a bad rap and is generally considered to be unreliable. I think the browsers local storage can be a viable option and it is reasonably stable and reliable, but, it is possible for the data to be wiped, which means for a lot of applications it's not going to be a great option. Even if it worked 99% of the time, that's still not good enough for storing most types of data.

In general, you should only use it where data loss would not be an issue, and it shouldn't ever be used to store sensitive data since it could potentially be accessed inappropriately. One example of where local storage might be a suitable option is if you wanted to store something like a temporary session token. This would allow you to tell if a user was already logged in or not, but if the data is lost it's not really a big deal because the user will just need to enter in their username and password again to re-authenticate.

If you are just using local storage to cache data from a server it would also not be a big

issue if the data is lost since it can just be fetched from the server again. Local storage is also great to use when we are just learning and building practice applications or prototypes because we don't really care if the data gets wiped.

Local Storage is a simple key-value system, and can be accessed through the globally available `localStorage` object:

```
localStorage.setItem('someSetting', 'off');

const someSetting = localStorage.getItem('someSetting');
```

This is the native (as in native to web browsers, not iOS or Android native) way to set and retrieve local storage data. **We won't be accessing local storage this way in Ionic applications**.

SQLite

SQLite is basically an embedded SQL database that can run on a mobile device. Unlike a normal SQL database, it does not need to run on a server and does not require any configuration. SQLite can be utilised by both iOS and Android applications (as well as others), but the SQLite database can only be accessed natively, so it is not accessible by default to HTML5 mobile apps.

We can, however, use Capacitor to easily gain access to this functionality. You would just need to add the SQLite plugin to your project:

```
npm install cordova-sqlite-storage
```

The main benefits of using SQLite are that it:

- Provides persistent data storage
- There is no size limitation on how much can be stored
- It provides the SQL syntax, so it is a very powerful tool for managing data

Although there are some differences in supported commands between SQL and SQLite, it is almost exactly the same. Here's an example of how you might execute some simple queries in SQLite:

```
var db = window.sqlitePlugin.openDatabase({name: "my.db"});

db.transaction(function(tx) {
  tx.executeSql('DROP TABLE IF EXISTS test_table');
  tx.executeSql('CREATE TABLE IF NOT EXISTS test_table (id integer
primary key, data text, data_num integer)');

  tx.executeSql("INSERT INTO test_table (data, data_num) VALUES
(?,?)", ["test", 100], function(tx, res) {
    console.log("insertId: " + res.insertId + " -- probably 1");
    console.log("rowsAffected: " + res.rowsAffected + " -- should
be 1");

  }, function(e) {
    console.log("ERROR: " + e.message);
```

```
  });
});
```

The example above looks pretty freaky, but if you're familiar with SQL then at least some of it should look familiar. This is the standard way to use SQLite with Cordova, but Ionic also provides its own service for using SQLite.

Although there are many people who do use SQLite directly like in the example above, most people only use SQLite as a way to provide a more permanent method for storing data, but still, treat it like a simple key/value store.

Ionic Storage

Fortunately, for most storage requirements we don't really need to worry about the implementation details. Ionic provides it's own storage service that will automatically make use of the best available storage mechanism (whether that is plain old local storage, Web SQL, IndexedDB or SQLite). It provides us with a consistent API to use no matter which storage mechanism is being used underneath.

To use it, all you have to do is install it:

```
npm install @ionic/storage
```

...and add the Ionic Storage Module to your **app.module.ts** file by importing it:

```
import { IonicStorageModule } from '@ionic/storage';
```

and adding it to the `imports` array:

```
  imports: [
    ...
    IonicStorageModule.forRoot()
    ...
  ],
```

Then you can just inject it into whatever class you need. Take this provider as an example:

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

@Injectable({
  providedIn: 'root'
})
export class Data {

  constructor(private storage: Storage){

  }
```

```
  getData(): Promise<any> {
    return this.storage.get('checklists');
  }

  save(data): void {
    this.storage.set('checklists', data);
  }

}
```

We simply call `this.storage.set` to store data on a particular key (in this case `checklists`) and then we can retrieve that same data later by accessing that key through `this.storage.get`. Keep in mind that when retrieving data from storage it will return a `Promise`, so in order to make use of the data you need to `await` it. For example, if I wanted to use the `getData()` function defined above I would need to do:

```
const data = await this.getData();
console.log(data); // This will contain the checklists
```

If your data storage requirements are more complex this may not be the perfect option, but for many scenarios, this provides a nice simple way to store the data we need. As long as we have the SQLite plugin installed, we can also be sure that our data isn't being saved somewhere that can easily be wiped out by the operating system.

Lesson 11: Fetching Data, Observables, and Promises

Although some mobile applications are completely self-contained (like calculators, soundboards, todo lists, photo apps, flashlight apps), many applications rely on pulling in data from an external source to function. Facebook has to pull in data for news feeds, Instagram the latest photos, weather apps the latest weather forecasts and so on.

In this lesson, we are going to cover how you can pull external data into your Ionic applications. Before we get into the specifics of how to retrieve some data from a server somewhere, I want to cover a little more theory on a few specific things that we are going to be making use of.

Mapping and Filtering Arrays

The **map** and **filter** functions are very powerful and allow you to do a lot with arrays. These are not fancy new ES6 or Angular features either, they've been a part of JavaScript for a while now.

To put it simply, **map** takes every value in an array, runs the value through some function which may change the value, and then places it into a new array. It **maps** every value in an array into a new array. To give you an example of where this might be useful, you might have an array of filenames like this:

```
['file1.jpg', 'file2.png', 'file3.png']
```

You could then map those values into a new array that contains the full path to the files:

```
['https://www.example.com/file1.jpg',
'https://www.example.com/file2.png',
'https://www.example.com/file2.png']
```

Doing that might look something like this:

```
const oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

const newArray = oldArray.map((entry) => {
  return 'https://www.example.com/' + entry;
});
```

So we supply the **map** with a function that returns the modified value. The function that we provide will have each value passed in as a parameter.

A **filter** is very similar to a **map**, but instead of mapping each value to a new array, it only adds values that meet certain criteria to the new array. Let's use the same example as before, but this time we want to return an array that only contains `.png` files. To do that, we would use a filter like this:

```
const oldArray = ['file1.jpg', 'file2.png', 'file3.png'];
```

```
const newArray = oldArray.filter((entry) => {
  return entry.indexOf('.png') > -1;
});
```

Suppose we still want to have the full path as well though. Fortunately, we can quite easily chain **filter** and **map** like this:

```
const oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

const newArray = oldArray.filter((entry) => {
  return entry.indexOf('.png') > -1;
}).map((entry) => {
  return 'http://www.example.com/' + entry;
});
```

Now we are first filtering out the results we don't want, and then mapping them to a new array with the full file path. The result will be an array containing the full paths of only the two `.png` files. I'm going to leave it there for now, but when we get into an example of how to fetch data soon it will become very clear why it was worth explaining how these work. Often we will want to work with data that isn't in the exact format we want it, and being able to manipulate data like this can make things a lot easier.

Observables and Promises

If you're reasonably experienced with JavaScript already, then you might be familiar with **Promises**, but far fewer people are familiar with **Observables**. Observables are one of the core features that are used in Angular applications (observables are provided by RxJS) so it is important to understand what they are and how they are different to Promises (they do look and behave *very* similar).

Before we get into Observables, let's recap what a Promise is at a very high level. Promises come into play when we are dealing with **asynchronous** code, which means that the code is not executed one line after another. In the case of making an HTTP request for data, we need to wait for that data to be returned, and since it might take 1-10 seconds for it to be returned, we don't want to pause our entire application whilst we wait. We want our application to keep running and accepting user input, and when the data from the HTTP request becomes available to us, *then* we do something with it.

A Promise handles this situation, and if you're familiar with callbacks it's basically the same idea, just a little nicer. Let's say we have a method called `getFromSlowServer()` that returns a promise, we might use it like this:

```
myMethod(){
getFromSlowServer().then((data) => {
  console.log(data);
});
}
```

or

```
async myMethod(){
  const data = await getFromSlowServer();
}
```

We call the `then` method which a Promise provides, which basically says "Once you have the data from the server, do this with it". If we use `async/await`, we are basically saying "pause this method to wait for the promise to resolve, then continue" - the important part with `async/await` is that the rest of the application still continues to work, it is just that one `async` method that will be "paused". In this case, we just get the data that the promise resolves with and we log it out. So our application will go about doing whatever else it has to do and when the data is available it will execute the code above. You could think of it like being at work and writing some report, you need some additional information so you ask your assistant to go find it for you, but you don't just sit there and wait for the assistant to get back - you keep writing your report and when the assistant returns *then* you use the information.

We understand what a Promise is now, so what's an Observable and what does it do that Promises don't?

An Observable serves the exact same purpose as a Promise, but it does some extra stuff too. The main difference between a Promise and an Observable is that a Promise returns a single result, but **an Observable is a stream that can emit more than one value over time**. It might be easier to think of Observables as **streams** because they are, but they are just called Observables because the stream is observable (as in, we can detect values that are emitted from the stream).

An Observable looks a lot like a Promise but, instead of using the `then` or `async/await` methods we used above, we use the `subscribe` method. Since a Promise only returns a single value, it makes sense to have that value returned and then do something. As I mentioned, an Observable is a stream that can emit multiple values, so it makes sense to subscribe to it (like your favourite YouTube channel) and run some code every time a value is emitted. It might look something like this:

```
someObservable.subscribe((result) => {
  console.log(result);
});
```

It is obvious that our program would need to wait for data to be returned when making an HTTP request, and thus Promises and Observables would be useful. It's not the only instance of where you will need to program asynchronously though. There are some less obvious situations like fetching locally stored data or even getting a photo from the user's camera, where you would also need to wait for the operation to finish before using the data.

If you want to go more in-depth into everything we've discussed above, I highly recommend this interactive tutorial. It introduces RxJS which includes Observables but also builds up a solid foundation of how to use **map**, **filter** and other functions. If you'd also like to dive into some more specifics about how an Observable differs from a Promise, I highly recommend this egghead.io video. These additional resources are not required in order to build Ionic applications, but they will help give you a deeper understanding.

Creating Your Own Observable

Generally, you don't need to create your own observables at a beginner level. We might make use of observables that prebuilt libraries return to us, but we won't often build our own observables. However, I would like to include a quick example here of how you might do that if you wanted to.

Probably the most common time we will make use of observables is when we are making HTTP requests to fetch data. This is an example of where a library will return an observable for us. We would call the `get` method on the HTTP library, and then `subscribe` to the **Observable** it returns. Since an Observable, unlike a Promise, is a stream of data and can emit multiple values over time, rather than just once, this concept isn't really demonstrated when using the **Http** service.

Let's pretend that we have a requirement where we need to listen for changes in a certain part of our application, and each time that happens we want to trigger a "save" function. We could use our own observable for this because the observable could emit some data every time a save needs to be triggered, and we could react to that.

When implementing this Observable you will see how to create an observable from scratch, and you'll also see how an Observer can emit more than one value over time.

Before we start creating this example, let's talk about Observables in a little more detail, in the context of what we're actually trying to do here. In the subscribe method in the code above we are only handling one response:

```
someObservable.subscribe((result) => {
  console.log(result);
});
```

which is actually the `onNext` response from the Observable. Observers also provide two other responses, `onError` and `onCompleted`, and we could handle all three of those if we wanted to:

```
someObservable.subscribe(

  (data) => {
    console.log(data);
  },

  (err) => {
    console.log(err);
  },

  () => {
    console.log("completed");
  }
);
```

In the code above the first event handler handles the `onNext` response, which basically means "when we detect the next bit of data emitted from the stream, do this". The second handler handles the `onError` response, which as you might have guessed will be triggered when an error occurs. The final handler handles the `onCompleted` event, which will trigger once the Observable has returned all of its data.

The most useful handler here is `onNext` and if we create our own observable, we can trigger that `onNext` response as many times as we need by calling the `next` method on the Observable and providing it some data.

Now that we have the theory out of the way, let's look at how to implement the Observable. We could set up a basic observable like this:

```
import { Injectable } from '@angular/core'
import { Observable, Observer } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class SomeService {

  public myObservable: Observable<any>;
  public myObserver: Observer<any>;

  constructor(){

    this.myObservable = Observable.create(observer => {
      this.myObserver = observer;
    });

  }

  makeSomethingHappen(): void {
```

```
    // stuff happens here
    this.myObserver.next(true);

  }

}
```

The first thing to notice here is that we are now importing **Observable** from the RxJS library. Then in our constructor, we set up the Observable:

```
    this.myObservable = Observable.create(observer => {
      this.myObserver = observer;
    });
```

Our `this.myObservable` member variable in the code above is now our very own observable. Since it is an observable, we can subscribe to it. We could subscribe to it within this class, or since this observable exists inside of a service we could inject that service elsewhere in the application and subscribe to it. For example, we could do the following (if the service were injected into a particular page in our application):

```
someService.myObservable.subscribe(data => {
  console.log(data);
});
```

This observable would be triggered whenever we call the `next` method on our `observer` in the service:

```
this.myObserver.next(true);
```

All we want to know in this case is that a change has occurred so we are just passing back a boolean (true or false), but we could also easily pass back some data if we wanted. There is a lot more to know about observables, but it is also one of those things that aren't really important to get a grasp on right away. For the most part, in the beginning, stages of Ionic development, you will only need to know how to subscribe to observables that are already created for you.

Using Http to Fetch Data from a Server

Ok, you should be armed with all the theory you need now - let's get into an example of fetching data using the HTTP library and observables. We're going to use the Reddit API to demonstrate here because it is publicly accessible and very easy to use. If you've purchased one of the packages for this book that includes the Giflist application then we will be exploring this in a lot more detail later.

You can create a JSON feed of posts from subreddits simply by visiting a URL in the following format:

https://www.reddit.com/r/gifs/top/.json?limit=10&sort=hot

If you click on that link, you will see a JSON feed containing 10 submissions from the **gifs**

subreddit, sorted by the hot filter. If you're not familiar with JSON, I would recommend reading up on it here – but essentially it stands for JavaScript Object Notation and is a great way to transmit data because it is very readable to humans, and is also easily parsed by computers. If you've ever created a JavaScript object like this:

```
const myObject = {
    name: 'bob',
    age: '43',
    hair: 'purple'
};
```

then you should be able to read a JSON feed pretty easily once you tidy it up a little. But how do we get it into our Ionic application?

The answer is to use the `HttpClient` service which is provided by Angular and allows you to make HTTP requests. If you're not familiar with what an HTTP request is, basically every time your browser tries to load anything (a document, image, a file etc.) it sends an HTTP request to do that. We can make an HTTP request to a page that spits out some JSON data, and pull that into our application.

First, we need to set up the `HttpClient` service, so let's take a look at a test page that has that service imported and injected into the constructor:

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

```
@Component({
  selector: 'app-page-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor(private http: HttpClient){

  }

}
```

This alone won't work, we also need to import the `HttpClientModule` into the root module (**app.module.ts**) for our application to make it available throughout the application:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";
import { HttpClientModule } from "@angular/common/http";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";
```

```
@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [
    BrowserModule,
    IonicModule.forRoot(),
    AppRoutingModule,
    HttpClientModule,
  ],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Since we have injected the `HttpClient` service into our constructor on the home page and made it available through `this.http` by using the `private` keyword, we can now make use of it anywhere in the home pages class.

Now let's take a look at how we might make a request to a reddit URL:

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

@Component({
  selector: 'app-page-home',
```

```
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss']
})
export class HomePage {

  constructor(private http: HttpClient){

      this.http.get('https://www.reddit.com/r/gifs/new/.json?
limit=10').subscribe(data => {
          console.log(data);
      });

  }

}
```

We just call the `get` method which returns an observable, we subscribe to that, and then when the observable emits data we log it out to the console.

IMPORTANT: Remember, HTTP requests are **asynchronous**. This means that your code will continue executing whilst the data is being fetched, which could take anywhere from a few milliseconds, to 10 seconds, to never. It's important that your application is designed to deal with this. To give you an example, if you were to run the following code:

```
this.posts = null;

this.http.get('https://www.reddit.com/r/gifs/top/.json?
```

```
limit=2&sort=hot')
.subscribe(data => {
    this.posts = data.data.children;
});

console.log(this.posts);
```

You would see null output to the console. But if you were to run:

```
this.posts = null;

this.http.get('https://www.reddit.com/r/gifs/top/.json?
limit=2&sort=hot')
.subscribe(data => {
    this.posts = data.data.children;
    console.log(this.posts);
});
```

You would see the posts output to the console because everything inside of the **subscribe** function will only run once the data has been returned. The first example runs `console.log` too quickly, and so the data is not ready when we attempt to log it out to the screen.

As I mentioned before, an Observable is useful because we can listen for multiple values over time... but why use it here? The Http call is only ever going to return one result, why

doesn't it just use a Promise instead of an Observable and save everyone the confusion?

The reason for a bit of favouritism of Observables over Promises is that an Observable can do everything a Promise can, it's technically better behind the scenes, and it can do extra fancy things that Promises can't. We could set up an **interval** for example so that the Http call fires every 5 or 10 seconds, we can easily set up **debouncing** which ensures that a request is not fired off too frequently (and subsequently making a ton of requests to a server), Observables can cancel old "in flight" requests if a new request is made before the result of the old request is returned and a whole bunch of other things.

Take this example from the Giflist application later in the book:

```
    this.subredditForm
      .get("subredditControl")
      .valueChanges.pipe(debounceTime(300),
distinctUntilChanged())
      .subscribe((subreddit: string) => {
        if (subreddit.length > 0) {
          this.redditService.reset(subreddit);
          this.contentArea.scrollToTop(200);
        }
      });
```

In this case, we get an observable using our form controls. The value of this can be controlled by the user using an input in the application. It's set up though so that the code inside the **subscribe** call will only run if there has been no change for more than 0.3

seconds (by using debounceTime) and it will also only run when a distinct value is supplied. This form control is attached to an input field that changes the subreddit. The reason we are doing this extra stuff is so that we aren't firing off unnecessary requests to change the subreddit (i.e. if the user is still typing, or if the value of the subreddit hasn't changed).

I think this example shows how a whole bunch of weird and useful stuff can be chained before the **subscribe** call to do a lot of useful things. We can essentially manipulate our observable in infinite ways to return different observables that we want to subscribe to.

If you're looking at the example above and thinking "Wow... Ionic is way too hard and confusing!" then don't worry too much. This is a pretty advanced example using Observables to demonstrate a point.

Observables are a huge topic, so there is a ton to learn. Don't feel intimidated if you don't really have much of an idea of what's going on though. Having a better understanding of Observables will help you when creating Ionic/Angular applications, but they are a reasonably small part of Ionic (well, they are a big part but you won't really have to deal with them much) and you can get by just fine by having just a basic understanding.

Fetching Data from your Own Server

We know how to pull in data using a JSON feed like the one provided by Reddit, but what if you want to pull in your own data? How can you go about setting up your own JSON feed?

Going into the detail of how to set up your own API is a bit beyond what I wanted to achieve with this lesson, but I would like to give you a high-level overview of how it's done.

Basically:

1. Make a request from your Ionic application to a URL on your server
2. Fetch the data using whatever server-side language you prefer
3. Output the required data to the page in JSON format

I'll quickly walk you through the steps of how you might implement a simple API with PHP, but you could use whatever language you want - as long as you can output some JSON to the browser.

1. Create a file called `feed.php` that is accessible at `https://www.mywebsite.com/api/feed.php`
2. Retrieve the data. In this case I'm doing that by querying a MySQL database but the data can come from anywhere:

```
$mysqli = new mysqli("localhost", "username", "password",
"database");
$query = "SELECT * FROM table";
$dbresult = $mysqli->query($query);

while($row = $dbresult->fetch_array(MYSQLI_ASSOC)){
    $data[] = array(
        'id' => $row['id'],
        'name' => $row['name']
    );
}
```

```
if($dbresult){
    $result = "{'success':true, 'data':" . json_encode($data) .
"}";
}
else {
    $result = "{'success':false}";
}
```

3. Output the JSON encoded data to the browser:

```
echo($result);
```

4. Use `https://www.mywebsite.com/api/feed.php` in your `this.http.get()` call in your application

As I mentioned, you can use whatever language and whatever data storage mechanism you like to do this. Just grab whatever data you need, get it in JSON format, and then output it to the browser. You don't even need a server, it could just be a static JSON file you are serving.

This should give you a pretty reasonable overview of how to fetch remote data using Ionic and the Http service. The syntax and concepts might be a little tricky to get your head around at first, but once you've got the basics working there's not really much more you need to know. Your data might get more complex and you might want to perform some fancier operations on it or display it in a different way, but the basic idea will remain the

same.

Lesson 12: Native Functionality

In the introductory section of this book, I introduced you to the concept of Capacitor. To recap, the main purpose of Capacitor is to:

1. Create a native application that contains a web view that runs our Ionic code. This allows the application to be submitted to native app stores.
2. Facilitate communication between the web view and Native APIs to allow for the integration of native functionality into Ionic applications.

We discussed the general idea behind Capacitor and its goals, as well as a simple example of getting it set up in a project. Now we are going to talk about how Capacitor is used in our projects in a little more depth. We will discuss actually creating and submitting the native builds with Capacitor in the **Building & Submitting** section later in this book.

Using Default Capacitor APIs

In the introduction section, we looked at a quick example of using the Camera API from Capacitor. The general idea is that you import the plugin into your class:

```
import { Plugins } from '@capacitor/core';
```

You create a reference to the particular plugin you want:

```
const { Camera } = Plugins;
```

and then you use the API as per the [documentation]():

```
Camera.getPhoto().then((photo) => {
  console.log(photo);
}, (err) => {
  console.log("Could not get photo");
});
```

or:

```
try {
  const photo = await Camera.getPhoto();
} catch(err) {
  console.log("Could not get photo");
}
```

In this example, as soon as this code is triggered the native Camera options will be launched for the user (allowing them to select a photo from their library, or take a new photo). This is of course not the only API that Capacitor offers. Capacitor provides a core set of APIs that allow you to access most "standard" functions of native devices. These

include:

- Accessibility
- App
- Background Task
- Browser
- Camera
- Clipboard
- Console
- Device
- Filesystem
- Geolocation
- Haptics
- Keyboard
- Modals
- Motion
- Network
- Share
- Splash Screen
- Status Bar
- Storage
- Toast

You can use any of these APIs by importing the core set of plugins from Capacitor, and creating a reference to them as we did for the Camera plugin:

```
const { Network } = Plugins;
```

Using Cordova Plugins with Capacitor

The core set of APIs will see you through a lot of circumstances, but there are often times when we want to integrate other kinds of native functionality. The types of features you may want to implement are pretty much endless, so the core set of APIs are not going to cover every conceivable idea you might come up with.

To deal with this, we can easily integrate 3rd party plugins (generally community maintained, but some companies also put out their own plugins) into our Capacitor applications. Capacitor is a reasonably new project, and so you won't find all that many 3rd party plugins available yet. Fortunately, Capacitor's predecessor "Cordova" already has a rich plugin ecosystem and community built up around it, and any Cordova plugin should be compatible with Capacitor.

To find plugins to include in your project, you will just be able to Google:

```
"(functionality you want) + Cordova plugin"
```

and you will more often than not find a suitable plugin. The issue with community-based plugins is that some are well designed and maintained, and others are not. So, you will have to investigate whether the project is currently being supported or not (e.g. see how much activity there is around the plugin when the last time it was updated was, and so on). A good way to find well-maintained plugins is to use those supported by Ionic Native (we will talk about what Ionic Native is in just a moment).

Using a Cordova plugin in a Capacitor project is as simple as adding the plugin through `npm`:

```
npm install cool-cordova-plugin
```

and then running:

```
npx cap sync
```

To copy the plugin over to your native project. Once it is installed, you can use the plugin as per the documentation for that plugin, often that might look something like this:

```
window.plugins.somePlugin.someMethod();
```

Keep in mind that if you use Cordova plugins in this way, your application may fail to compile due to TypeScript warnings. This is because TypeScript does not know what it is, and you may need to install **typings** for it. This is typically done by running:

```
npm install @types/name-of-package-you-are-using
```

or to brute force your way past this, you can simply add:

```
declare var variableCausingProblems;
```

above the decorator in the class that you are using the plugin in. Another (better) way to integrate Cordova plugins is to use the Ionic Native API for it if it is available.

Ionic Native

As we have discussed so far, we can use the default Capacitor APIs in our project, or we could install Cordova plugins and use those directly as well. However, there is also another option available to us.

Ionic Native "wraps" plugins so that they behave more nicely in an Ionic application. This doesn't change the functionality, but the default way in which we access a Cordova plugin (generally by using global objects) breaks the typical style for an Ionic/Angular application. We usually rely on injecting dependencies into our classes and promises/observables rather than using global objects with callbacks.

If there is an Ionic Native wrapper available for a particular plugin, we can install that Ionic Native package and use that to interface with the plugin. I would recommend doing this whenever Ionic Native supports a particular plugin. As I mentioned, it's also a good way to find Cordova plugins that are well supported/maintained.

Ionic Native is installed by default in all Ionic applications, so all you need to do is install the plugin you want to use, just like you would normally, for example:

```
npm install cordova-plugin-geolocation
```

and you will also need to install the package for the plugin from Ionic Native, i.e:

```
npm install @ionic-native/geolocation
```

Once the plugin is installed, you will need to add a provider for it in the **app.module.ts** file by importing it:

```
import { Geolocation } from "@ionic-native/geolocation/ngx";
```

and adding it to the `providers` array:

```
  providers: [
    ...
    Geolocation,
    ...
  ]
```

Next, you will need to import the plugin from Ionic Native into the class you want to use it in:

```
import { Geolocation } from "@ionic-native/geolocation/ngx";
```

and finally, you will need to inject the plugin into the constructor in the class you want to use it in:

```
constructor(private geolocation: Geolocation){

}
```

and then you can use it in your code:

```
this.geolocation.getCurrentPosition().then((resp) => {
  console.log("Latitude: ", resp.coords.latitude);
  console.log("Longitude: ", resp.coords.longitude);
});
```

or:

```
const resp = await this.geolocation.getCurrentPosition();
```

Notice that in the code above a promise is returned and we set up a handler using `.then()`, if we were just using the standard Cordova syntax this wouldn't be possible - we would instead have to use callback functions, which are a bit messier.

We are using the Geolocation plugin as an example here, but keep in mind that **this is already available as an API through Capacitor** so we would just use the Capacitor API in this instance, e.g:

```
const { Geolocation } = Plugins;
```

```
const position = await Geolocation.getCurrentPosition();
```

When there is a Capacitor plugin and a Cordova plugin available that do the same thing, you should give preference to using the Capacitor plugin.

It's also important to note that not all Cordova plugins are available in Ionic Native. For a list of all of the available plugins, and how to use them, you should check the Ionic Native documentation. If a plugin you want to use is not available in Ionic Native, then you can just go back to using the standard Cordova syntax (or you can add it to Ionic Native yourself).

Modifying Native Project Files

Capacitor creates native project files for each platform you are building for, and these are no different to any standard native project. Capacitor literally creates a standard native project, adds a web view to it, and loads your Ionic code into that web view (plus some extra Capacitor stuff we can make use of to communicate with Native APIs). The files that Capacitor will create for iOS will look just like any other standard iOS native project, and

the same goes for Android. This means that you can also modify these native files however you wish, you aren't restricted to just making changes to the web code when building Ionic applications. This isn't generally required, but it's important to know that it is possible.

Sometimes, plugins will require that you make changes to the native **Info.plist** file for iOS, or perhaps the **AndroidManifest.xml** file for Android. This is uncommon, but it is sometimes required.

You aren't limited to just modifying these configuration files, though, you can modify anything you like.

Custom Plugins

We discussed that Capacitor provides a default set of APIs and that there are also community plugins available. Since you can incorporate any community plugin into your project, that means you can also create your own!

Capacitor makes this process reasonably straight-forward, but it will require more advanced knowledge and it requires writing native code (I expect most beginners wouldn't feel comfortable developing their own Capacitor plugins).

This is out of scope for this book, however, I have filmed a tutorial on creating your own Capacitor plugins if you want to check it out. It is rare that you wouldn't be able to find a plugin that already exists to do what you want, but in the case that it does, you can take matters into your own hands.

Quick Lists

Lesson 1: Quick Lists Introduction

Quick Lists is the de facto step-by-step tutorial application for this book - no matter which of the packages you purchased, you will have access to this application. The reason I chose Quick Lists to fill this role is that it covers a broad range of the core concepts in Ionic, and the skills you learn throughout building this application will be used frequently in most other applications you create.

A lot of people (myself included) create todo application tutorials when explaining some new technology or framework, the reason for this is usually because a todo application covers most of the basic things you would want to do in an application, for example:

- General structure & setup
- User interface
- Creating, reading, updating and deleting data
- Accepting user input

These are all obviously important concepts that need to be covered, but I *really* wanted to avoid building another todo application for this book - I wanted to do something that was just a little bit more complex and interesting - to start getting exposed to some of the more tricky situations you run into in real-world situations. The result is pretty similar and covers the same bases as a todo application would, but I think it's a little more fun to build and cranks up the complexity just a little bit.

About Quick Lists

The idea for Quick Lists came from a personal need of mine. At the time of writing this originally I was working remotely and traveling around Australia in a caravan (I think people in other parts of the world would refer to this as a "trailer" or "camper trailer" or "motor home"). It's certainly a great experience, but essentially lugging your entire house around the country (and it's a big country) every week or so comes with some complications.

One particularly complicated thing is packing up and hitching the caravan to the car, as well as unhitching the caravan and setting it up. I won't bore you with the details, but there are at least 20 or so different things that need to be done and checked each time. Some are inconsequential, but some are really important like making sure the chains are attached securely to the car, that the gas is off, and that the brakes and indicators are working.

I decided to create an app where you could create "pre-flight" checklists. The checklists would contain a bunch of items that you could check off as being done - a *repeatable* todo list application in a sense.

As I mentioned, this application covers similar concepts to what a traditional todo application would. Specifically, the main concepts we will cover are:

- Complex Lists
- Data Types / Interfaces
- Forms and User Input
- Simple Navigation
- Passing Data Between Pages
- Creating, Reading, Updating and Deleting Data
- Data Storage and Retrieval

- Theme and Styling

Here's a quick rundown of the exact features of the application:

- The first time the user uses the application an introduction tutorial will be shown
- The user can create any number of checklists
- The user can add any number of individual items to any checklist
- An item in a checklist can be marked as complete or incomplete
- The user can edit or delete any checklist or items in a checklist
- The user can "reset" the completion state of an entire checklist with one action
- All data will be remembered upon returning to the application (including the completion state of checklist items)

and here are a couple of screenshots to put everything in context.

Your lists
Caravan Hitching
9 items
4 items
Long Travel
6 items
Winter Pruning
4 items
Tax
3 items

Back
Caravan Hitching
Remove locks
Raise legs
Remove power cord
Remove drinking hose
Remove sullage hose
Secure annexe
Turn off gas
Store jockey wheel
Connect u-bolts

Lesson Structure

1. Getting Ready
2. Basic Layout
3. Creating an Interface and Checklist Service
4. Creating Checklists and Checklist Items
5. Saving and Loading Data
6. Creating an Introduction Slider & Theme

Ready?

Now that you know what you're in for, let's get to building it!

Lesson 2: Quick Lists Getting Ready

In this lesson, we are going to prepare our application for the journey ahead. We are going to generate the application, and we are also going to set up all of the components, plugins, and routes that we need. The idea is to get all of the general scaffolding required for most projects out of the way so we don't have to mess around with creating files and configuring things throughout the rest of the lessons.

At the end of this first lesson, we should have a nice skeleton application set up with everything we need to start diving into coding.

A good rule of thumb before starting any new application is to make sure you have the latest version of the Ionic CLI installed, so if you haven't done it recently then make sure to run:

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

If your environment is not set up correctly, you may run into permission errors when installing this package globally. If you are having permission issues, you can use the Ionic teams guide to solving these issues: Resolving permission errors.

If you do not already have a recent version of Node/npm set up on your machine, make sure to complete the Environment Setup guide first.

Generate a new application

We will be using the blank starter template for this application which, as the name implies, is basically an empty Ionic project. It comes with one page built in called **HomePage** which we will reuse as our main page that will hold our checklists in the next lesson.

> Run the following command to generate a new application

```
ionic start quicklists blank --type angular
```

When you are prompted with the following:

```
? Integrate your new app with Capacitor to target native iOS and
Android? (y/N)
```

Say **Yes**, as we will be utilising Capacitor for native builds. If you do not set up Capacitor when first creating your application, you can always do it manually later as well by running the following command:

```
ionic integrations enable capacitor
```

NOTE: You should still install Capacitor in this project even if you do not intend to build for iOS or Android, as we will still be using Capacitor for the web platform as well.

> Make the new project your current working directory by running the following command:

```
cd quicklists
```

Your project should now be generated and you can open up the project folder in your favourite editor (I recommend using Visual Studio Code). You can take a look at the application as it is now by running the following command:

```
ionic serve
```

which for now should look something like this:

Blank

Ready to create an app?

Start with Ionic UI Components

Create the Required Components

This application will have a total of three "page" components. We will have our **HomePage** that will display a list of all the checklists, an **IntroPage** that will display the introduction tutorial, and a **ChecklistPage** which will display the individual items for a specific checklist. We've already got the Home page, so let's create the other two now.

NOTE: You will need to stop serving your application in order to run these commands. If your application is currently being served through the command line, make sure to hit `Ctrl + C` first to stop the process.

> Run the following command to generate the Introduction page:

```
ionic g page Intro
```

> Run the following command to generate the Checklist detail page:

```
ionic g page Checklist
```

Create the Required Services

We are going to be creating a **service/provider** in this application to help us out. This service will handle saving the checklist data into storage, and also making the checklist data available throughout the entire application.

> Run the following command to generate a Checklist service:

```
ionic g service services/Checklist
```

NOTE: Notice that we use `services/Checklist` instead of just `Checklist` this time. This is because we want to generate all of our services inside of a folder called `services`. You could also use this same technique for pages if you wanted to store all of your pages inside of a specific folder like `pages`.

Create the Interfaces

We will eventually be defining a custom interface to represent our `Checklist` and `ChecklistItem` data types. There is no generate command for this, so you will need to create it manually.

> Create a new folder inside of the `app` folder called `interfaces`

> Create a new file inside of the interfaces folder called `checklist.ts`

> Create a new file inside of the interfaces folder called `checklist-item.ts`

Configure the Routes

As we discussed in the basics section, we need to define routes so that our application can match up the current URL to a particular component that you want to display. Generating the application and using the generate commands do most of the work for us, as routes are automatically injected into the **app-routing.module.ts** file. However, we will need to make some slight modifications to these.

> **Modify `src/app/app-routing.module.ts` to reflect the following:**

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

const routes: Routes = [
  {
    path: "",
    redirectTo: "/checklists",
    pathMatch: "full",
  },
  {
    path: "intro",
    loadChildren: () => import("./intro/intro.module").then((m) =>
m.IntroPageModule),
  },
  {
    path: "checklists",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
  },

  {
    path: "checklists/:id",
    loadChildren: () =>
import("./checklist/checklist.module").then((m) =>
```

```
m.ChecklistPageModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

We have mostly kept the routes the same - we just have one route linking up to each page in our application - however, we have also added a route parameter of `id` to the last route so that we can pass an `id` into our checklist page. This is what will allow us to select a checklist on the home page, and then pass it to the checklist page where we can see more details about it. We will just need to pass the `id`, and then we will use that `id` to look up the actual checklist object in our checklist service.

Configure Capacitor

At this point, you should already have Capacitor added to your project. However, if you did not respond **yes** to the prompt about installing Capacitor when initially creating the project, you should manually set it up now.

> Run the following command (if required) to enable Capacitor:

```
ionic integrations enable capacitor
```

Keep in mind that by default your application will be given a **Bundle ID** of `io.ionic.starter`. This Bundle ID is used to identify your native iOS/Android builds, and you will eventually want to change this to something unique to you or your company. You can change it later if you wish, but it is easier to do before we add the iOS and Android platforms. You may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`. If you don't intend on eventually modifying and publishing this project, then this step doesn't matter.

If you look at the **capacitor.config.json** file, you will notice that the `webDir` is listed as `www`. This is the folder that Capacitor will look to when it is copying over the code for your application into the native projects. It doesn't care about the rest of your source code, it only cares about the built output (i.e. the code that is actually run through the browser). By default, the `www` folder will not exist in your project until you perform a build of your application manually. We are going to do that now.

> Run the following command to generate a build of your application:

```
ionic build
```

It is good to use the `ionic build` command whenever you want to test your application on a device during development, but keep in mind that this creates a development build.

When you are building the final version of your application, or if you want to test a production version of your application, you should run:

```
ionic build --prod
```

which will create an optimised production build.

Add Native Platforms

By default, the iOS and Android platforms will not be enabled in your project. If you want to add these platforms you can do so now (or you can do it later if you prefer). Remember, building for iOS will require Xcode and building for Android will require Android Studio - if you want to add these platforms now, you should make sure you have followed the instructions in the Generating an Ionic Application lesson to install Xcode, Android Studio, and their dependencies.

```
ionic cap add ios
```

Keep in mind that the first time the `Updating iOS native dependencies` line is run it may take quite a while.

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your Ionic application code and dependencies at any time by running:

```
ionic cap sync
```

IMPORTANT: If you want to use a production build with Capacitor, you will need to add the `--prod` flag to the sync command, e.g: `ionic cap sync --prod`.

When you make changes to your application build, you will need to run this command to copy the changes over to the native iOS/Android projects created by Capacitor. If you do not run this command, any changes you make to your Ionic application will not be present when running the application on iOS/Android.

Set up the Root Component

The default starter templates for Ionic utilise **Ionic Native** to handle hiding the **Splash Screen** and styling the **Status Bar**. Whilst you can still use **Ionic Native** and **Cordova** plugins inside of a Capacitor project, Capacitor provides this functionality through default APIs, so we are going to use those instead. It is also important to remember that the Cordova Splash Screen plugin is not compatible with Capacitor, so make sure that you do not install it in your project.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor() {
    this.initializeApp();
  }

  async initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#2dd36f" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

You can also use the `StatusBar` API to hide the status bar entirely, but in this case we are styling it with a colour that will match our application. The status bar is the bar at the top of native applications that display the time, battery level, etc.

Set up Ionic Storage

We will be making use of the Ionic Storage API in this application. Ionic provides a simple key/value storage API that we can use in our applications to store data - it provides a consistent API and will automatically use the best storage mechanism available. That means that if we install the SQLite plugin in our project, it will store our data in native storage (rather than browser local storage which is more likely to get wiped by the operating system randomly). If native storage is not available, it will default back to browser storage.

To enable Ionic storage, you need to install it.

> Install Ionic Storage with the following command:

```
npm install @ionic/storage
```

and you will also need to add it to your root module file.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported `IonicStorageModule` and we have also added it to the `imports` array. We have also removed the `SplashScreen` and `StatusBar` providers from Ionic Native since we are not using them anymore.

Set up Plugins

We are going to set up any native plugins that we need now. Remember that when installing Cordova plugins in a Capacitor project you should install them using the `npm install` command, **not** with `ionic cordova plugin add`.

> Run the following command to add the SQLite plugin:

```
npm install cordova-sqlite-storage
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

Remember, after making changes to the native plugins in your project you will need to run:

```
ionic cap sync
```

This will copy the new plugins you have installed over to the native projects.

Set up Images

When building this application we are going to be making use of a few images. I've included these in your download pack but you will need to set them up in the application you generate.

> Copy the images folder in the download pack for this application from `src/assets` to your own `src/assets` folder

Summary

That's it! We're all set up and ready to go, now we can start working on the interesting stuff. Make sure to test your application by running `ionic serve` every now and then, it's easier to catch and fix errors along the way than waiting until later. If you ever get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again.

Lesson 3: Basic Layout

We're going to start things off pretty slow and easy in this lesson, and just focus on creating the basic layout for the application. We will need to create templates for the **Home** page, which will display all of the checklists that have been created, and the **Checklist** page, which will display all of the items for one specific checklist. If you've been paying attention then you'll know there's also one more page, but we will be creating that later.

As I've mentioned before, I've tried to make this book as modular as possible so that you can build the applications that interest you first, and so that you aren't forced to follow a particular order. However, since this is the first application, and because it is the only application contained in the basic package, I'll be paying special attention to making sure all the little things are explained thoroughly.

The Home Page

Before we jump into the code, let's get a clear picture in our mind of what we are actually building. Here's a screenshot of the completed homepage:

Your lists
Caravan Hitching
9 items
4 items
Long Travel
6 items
Winter Pruning
4 items
Tax
3 items

It has got a bit of fancy styling which we will cover later, but essentially it's a pretty simple list of items with a button in the top-right to add a new checklist. It's not entirely simple though, you'll notice one of the list items looks a little different and is displaying an **Edit** and **Delete** button. This is because we will be using the `<ion-item-sliding>` component that Ionic provides, which allows us to display some additional content (typically action buttons) that will be displayed when the user swipes a particular list item.

Let's get into building it. First, we're going to look at the entire template to see everything in context, then we're going to break it down into smaller chunks that we will discuss in detail.

> Modify `src/app/home/home.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="success">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="addChecklist()">
        <ion-icon slot="icon-only" name="add"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

```
<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title color="light" size="large">Your lists</ion-title>
    </ion-toolbar>
  </ion-header>

  <div id="container">
    <ion-list lines="none">
      <ion-item-sliding>
        <ion-item routerLink="/checklists/123"
routerDirection="forward">
          <ion-label>TITLE GOES HERE</ion-label>
          <span>0 items</span>
        </ion-item>

        <ion-item-options side="end">
          <ion-item-option color="light"
(click)="renameChecklist(checklist)">
            <ion-icon slot="icon-only" name="clipboard"></ion-
icon>
          </ion-item-option>
          <ion-item-option color="danger"
(click)="removeChecklist(checklist)">
            <ion-icon slot="icon-only" name="trash"></ion-icon>
          </ion-item-option>
        </ion-item-options>
      </ion-item-sliding>
```

```
    </ion-list>
  </div>
</ion-content>
```

Let's start off by discussing the `<ion-header>` section:

```
<ion-header>
  <ion-toolbar color="success">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="addChecklist()">
        <ion-icon slot="icon-only" name="add"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

The `<ion-toolbar>` allows us to add a header bar to the top of our application that can hold buttons, titles, and we will also often use this area to display a back button for navigation.

We add the `color` property to the toolbar to style it with our `success` colour, this is

defined by default but we can also modify it to be whatever we like (we covered this in the **Styling & Themes** lesson earlier in the book, and we will be modifying these theme values for this application later). Inside of the toolbar component we use `<ion-title>`, which is typically used to display a text title for the current page, to display our logo. We also use `<ion-buttons>` component to allow us to position an `<ion-button>` nicely within our toolbar. By using the `end` slot, the buttons will be placed on the right-hand side of the toolbar, with the position and styling that you would typically expect of an action button in a mobile application toolbar.

Finally, we have the button itself inside of `<ion-buttons>`. This button uses an `add` icon and has a click handler attached to it, which will call the `addChecklist()` function in our **home.page.ts** file (which we have not implemented just yet). Also, notice that we use the `icon-only` slot for the icon, this will make the `<ion-icon>` component style the icon so that it is bigger (if it were being displayed alongside text, then we would want the icon to be smaller).

It is also worth noting that we have a second header section *inside* of our `<ion-content>` area:

```
<ion-header collapse="condense">
  <ion-toolbar>
    <ion-title color="light" size="large">Your lists</ion-title>
  </ion-toolbar>
</ion-header>
```

This is to achieve the effect on iOS where we have a large header title sitting in the content area, and then when the list is scrolled down the title will shrink down and be displayed in the toolbar instead. This is included by default in our generated application, but you can remove this section if you like to get rid of the effect. When viewing the page on an Android device, this will have no effect.

Let's move onto the content and list section now:

```
<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title color="light" size="large">Your lists</ion-title>
    </ion-toolbar>
  </ion-header>

  <div id="container">
    <ion-list lines="none">
      <ion-item-sliding>
        <ion-item routerLink="/checklists/123"
routerDirection="forward">
          <ion-label>TITLE GOES HERE</ion-label>
          <span>0 items</span>
        </ion-item>

        <ion-item-options side="end">
          <ion-item-option color="light"
(click)="renameChecklist(checklist)">
```

```
            <ion-icon slot="icon-only" name="clipboard"></ion-
icon>
          </ion-item-option>
          <ion-item-option color="danger"
(click)="removeChecklist(checklist)">
            <ion-icon slot="icon-only" name="trash"></ion-icon>
          </ion-item-option>
        </ion-item-options>
      </ion-item-sliding>
    </ion-list>
  </div>
</ion-content>
```

Before we get to the list itself, notice that everything is wrapped inside of `<ion-content>` - this is what holds the main content for the page and in most cases, everything except the header section with the toolbar will be inside of here. It is also worth noting here that we have set the `fullscreen` property to `true`. This will cause the content area to scroll *behind* the toolbar, rather than the content area being cut off where the toolbar is. Often, this isn't required, but in order to achieve the iOS effect that we just discussed it is important to set the `fullscreen` property.

Now let's focus on the `<ion-list>`. Much like a list is created with plain HTML, e.g:

```
<ul>
  <li></li>
```

```
  <li></li>
  <li></li>
</ul>
```

A list in Ionic is created in basically the same way:

```
<ion-list>
  <ion-item></ion-item>
  <ion-item></ion-item>
  <ion-item></ion-item>
</ion-list>
```

Of course, ours looks a little more complicated than that so let's talk through it. The first thing out of the ordinary we are doing is setting the `lines` property on the `<ion-list>` to `none`. Just like the `secondary` property, we added to the toolbar, this property also styles our list except that it will cause the items in the list to not display with borders.

The next bit gets a little trickier, as it is where we set up our sliding item. An `<ion-item-sliding>`, opposed to a normal `<ion-item>`, has two sets of content - the item itself, and then the `<ion-item-options>` which will be revealed when the user slides the item.

The first block of code inside of `<ion-item-sliding>` is the normal `<ion-item>` component. We have also attached a `routerLink` to our `<ion-item>` which, if you remember from the section on navigation, allows us to link to a route in our application. In

this case, we are going to the `checklists` route, and we are also passing the `id` of the checklist through the URL. This will eventually allow us to show the details for a specific checklist, but for now, we are just passing a dummy `id` of `123`. Eventually, we will be creating a bunch of these items from an array of data, and we will create a reference to each individual item so that we can link to each item's own `id`.

Finally, we have the second block of code, `<ion-item-options>`, which simply allows us to define what content we want to display when the user slides the item. In this case, we are just adding edit and delete buttons which will call a function and also pass in a reference to the checklist it was called on (this will allow our code to figure out which checklist it needs to delete or edit). Again, we will have to create this reference later, for now... clicking on it will cause errors.

That's all there is to the home page, so let's move on to the checklist page.

The Checklist Page

As we did before, let's first take a look at what we are building before we jump in:

Back
Caravan Hitching
Remove locks
Raise legs
Remove power cord
Remove drinking hose
age hose
Secure annexe
Turn off gas
Store jockey wheel
Connect u-bolts

This screen looks very similar to the last one, and for the most part, it is, but there are some differences. Obviously, we have an extra button now, and a back button to return to the main page. The items in our list also now have a checkbox next to them that will be used for marking an item as complete, and we still have our sliding items set up.

Again, let's add the code for the template to the application and then talk through it:

> Modify `src/app/checklist/checklist.page.html` to reflect the following

```
<ion-header class="ion-no-border">
  <ion-toolbar color="success">
    <ion-title>CHECKLIST TITLE</ion-title>
    <ion-buttons slot="start">
      <ion-back-button color="light" defaultHref="/checklists">
</ion-back-button>
    </ion-buttons>
    <ion-buttons slot="end">
      <ion-button (click)="restartList()">
        <ion-icon slot="icon-only" name="refresh"></ion-icon>
      </ion-button>
      <ion-button (click)="addItem()">
        <ion-icon slot="icon-only" name="add"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

```
<ion-content>
  <ion-list lines="none">
    <ion-item-sliding *ngFor="let item of checklist.items">
      <ion-item color="success">
        <ion-label>ITEM TITLE</ion-label>
        <ion-checkbox
          color="light"
          [checked]="item.checked"
          (ionChange)="toggleItem(item)"
        ></ion-checkbox>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="light" (click)="renameItem(item)">
          <ion-icon slot="icon-only" name="clipboard"></ion-icon>
        </ion-item-option>
        <ion-item-option color="danger"
(click)="removeItem(item)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

You already know how the toolbar works, but we have changed a couple of things. We

added a button in a separate `start` slot. This is our back button which will allow us to navigate back to the page we came from (i.e. the **Home** page that contains a list of all of the checklists). We also supply a `defaultHref`, this way if the browser is reloaded (and the history is lost), or the user navigates directly to this page when they first load the application by supplying the URL to this page specifically, the back button will still know what page it needs to navigate the user back to. It is also worth noting that we are using one of Ionic's CSS Utility classes on the `<ion-header>`: `ion-no-border`. This will remove the border on the header for us automatically - we could do this ourselves with CSS as well, but Ionic provides a bunch of these types of classes to make our job easier.

We are using the `<ion-title>` in a more traditional way this time to display the title of the checklist that is currently being viewed (at least, we will be doing that soon). The buttons also both have click handlers to different functions, but since we are in the **ChecklistPage** component now, these functions will be triggered in the **checklist.page.ts** file (which we also have not created yet).

You also know how the sliding items work, but this time we have an `<ion-checkbox>` inside of our main `<ion-item>` and when that checkbox has its state changed it will trigger the `toggleItem()` function which we will need to define later. Notice that we are listening for the `ionChange` event on the checkbox so that we can detect when the state of the checkbox changes.

There's also a bit of new syntax here, let's take a closer look:

```
[checked]="item.checked"
```

When something is surrounded by `[square brackets]` it means we will be binding to a **property** on that element, and we will be setting it to the **expression** contained in the quotes, not a string. So, in this case, it would set the **checked** property to the value of whatever **item.checked** evaluates to be. Right now we haven't created a reference to **item** so it won't work anyway, but later `item` will reference the specific item that is being used, and we will be able to determine its `checked` status. The important thing to remember here is that the square brackets will evaluate whatever is inside the quotation marks. Let's imagine you have the following defined in the class for your component:

```
this.myName = "Josh"
```

If I were to use the following code:

```
<something myName="myName">
```

the `myName` **attribute** would be set to literally "myName", but if I were to use this code instead:

```
<something [myName]="myName">
```

the myName **property** would be set to `Josh` because `myName` would get evaluated first in this instance, rather than being treated like a string. Moving on, there's nothing else surprising in the rest of the template - we simply add the 'Edit' and 'Delete' buttons to our sliding list like we did on the last page.

If you run `ionic serve` now, you should see something like this (assuming you have Chrome DevTools set to use an iPhone simulator - if you are viewing the page without a

simulator, or if you are using an Android device simulator, it will look a little different):

Your lists

TITLE GOES HERE 0 items >

I think we can both agree that it is pretty ugly at the moment, but the structure is there.

In the following lessons, we will be getting the list to pull in real data and styling it so that it looks a lot better. The template syntax in Ionic may look a little confusing at first, but once you get your head around it, it's quite nice to use (and it certainly beats building out all of your own components for everything).

Lesson 4: Creating an Interface and Checklist Service

We have the majority of our template for the application set up which handles the *presentation* for the application, but we are also going to need to handle the *logic* that needs to occur in order for the application to function as intended. We have a page to display our checklists (which don't exist yet), and a page to display the details/items for a particular checklist, but now we need to start working on how to actually make things *happen* on those pages.

We will need a way to create checklists, create items to add to the checklists, and we will need a way to modify (update and delete) that data. This is what we will be focusing on in this lesson.

Whenever you have some kind of **entity** in your application - things like **articles**, **users**, **products**, or in this case **checklists** - it's generally a good idea to create a **service** that handles all of the operations related to that entity. In this case, we would use our service to:

- Create new checklists
- Add items to a checklist
- Update a checklist,
- Change the completion state of an item in a checklist
- Delete an item in a checklist
- ...and so on

A service is not all that different to a regular page/component, but its purpose is purely to

run logic - it doesn't have a template to display things directly to the user. Unlike a page/component that becomes "active" when we go to a certain route, a service is just a "helper" that is used inside of other pages/components. We can even use the same service in multiple different pages to access the same set of functionality or share data between pages. By default, a service in Angular will be a **singleton** which means a single instance of the service will be shared throughout the application no matter where you are using it, which makes it a great way to share data throughout the application.

We don't *need* to create services to handle all of the logic in our application, because each page that we have also has its own class that we can create functions in. We could, for example, handle all of the logic for creating and updating checklists inside of our **home.page.ts** file. However, it's better to get pages to do as little "work" as possible. A page should ideally contain simple logic that is specific to *only* that page. For example, if we have a `(click)` listener attached to a button in the home page's template, then it makes sense to create some logic for that in the **home.page.ts** file (e.g. the method that gets triggered when the button is clicked). However, any kind of complex logic, or logic that relates to "entities" in your application, should generally happen inside of a "service" that is injected into that page. If as a result of clicking that button we wanted to trigger some kind of complex operation, we would still have a method in our **home.page.ts** file to handle the click event, but then that method would likely make a call to some service to handle the additional "work" (rather than the page doing everything itself). Talking about this in the abstract might be a bit confusing, but you will see examples of this as we progress.

Also consider that you might want to do something like edit an article title in an application. You might have two different pages in your application that allow a user to edit the title of the article. Rather than implementing this logic twice - once for each page - we can just implement it once in a service, and have both pages call a method on that service to edit

the article, e.g: `this.articleService.updateArticleTitle(5, "BREAKING: Ionic releases a new component")`.

Services may also be referred to as **providers** or **injectables**. We use the `@Injectable` decorator to create a service with Angular, and the service that we create can be "injected" into any of the other pages/components in our application through the constructor, e.g:

```
import { MyService } from '../../services/myservice.service';

//...snip...

constructor(private myService: MyService){

}
```

After injecting `MyService` as `myService` in the example above, we would then be able to access the functionality that `MyService` provides anywhere within that class like this:

```
this.myService.someMethod();
```

Create an Interface

Before we create the service that will handle everything related to checklists in our application, we are first going to create something called an **interface**.

A **service** will allow us to handle operations related to checklists in our application, and an **interface** will allow us to define what a checklist *is*.

We've talked previously about using "types" in our Ionic applications, e.g:

```
let myString: string = "hello";
```

In this example, we have given `myString` a type of `string`. Although this type is not strictly required for the operation of the application, TypeScript will now warn us if we ever attempt to set `myString` to anything that is not a string.

Now, let's say that a checklist in our application would be an object that looks like this:

```
let myChecklist = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      title: 'Item One',
      checked: false
    },
    {
      title: 'Item Two',
      checked: true
    }
```

```
  ]
};
```

Seeing the value of TypeScript and types, we may consider this fantastic object of ours and decide: "I shall give this an appropriate type!".

We could just give it a type of `Object`, because it *is* an object:

```
let myChecklist: Object = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      id: '1',
      title: 'Item One',
      checked: false
    },
    {
      id: '2',
      title: 'Item Two',
      checked: true
    }
  ]
};
```

However, an `Object` is a pretty generic description for something that may follow a very strict format. In this case, our checklists are always going to have an `id` that is a string, a `title` that is a string, and a `items` property that will be an `array`. But a type of `Object` will only enforce that it is any kind of object.

To enforce this structure, we can create our own custom type with an **interface**.

This isn't strictly required, in fact, you don't *have* to give types to your data at all, but it's a good habit to get into. Adding appropriate types will keep the TypeScript compiler happy, and it will allow you to catch a lot of bugs/issues in your application before they ever become issues.

> Modify `src/app/interfaces/checklist-item.ts` to reflect the following:

```
export interface ChecklistItem {
  id: string;
  title: string;
  checked: boolean;
}
```

> Modify `src/app/interfaces/checklist.ts` to reflect the following:

```
import { ChecklistItem } from "./checklist-item";

export interface Checklist {
```

```
  id: string;
  title: string;
  items: ChecklistItem[];
}
```

To create an interface, all we need to do is supply each property the object expects, and we need to give each property the appropriate type. Notice that for the `items` property of the `Checklist` interface that we give it a type of `ChecklistItem[]` which means: an array of elements that have a type of `ChecklistItem`. This `ChecklistItem` type is another interface we have defined, which describes the items that will be added to the checklist.

Later, when we are working with our checklists, we will be able to import these interfaces and use them as our own custom types:

```
const myChecklist: Checklist = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      id: '1',
      title: 'Item One',
      checked: false
    },
    {
```

```
      id: '2',
      title: 'Item Two',
      checked: true
    }
  ]
};
```

The more specific you get with your types the better... but I wouldn't blame you for just going with the more generic `Object` type here if you are still getting used to types. You can even use the `any` type to completely circumvent needing a type at all - this is also the most useless type, though, because it will allow absolutely any kind of data and basically circumvents to whole point of using types.

Some people probably wouldn't be too happy at me suggesting that, but I think it's fine to give yourself a bit of breathing room as a beginner and not worry too much with finicky best practice type things. If you are working on a serious production application, then I would implore you to take the time to set up interfaces like this. If you are just messing around and trying to learn, I don't think it's that important. Pick one or a few things at a time that you can improve upon in your code, rather than trying to learn and do everything at once.

Create the Checklist Service

Now we are going to create our checklist service. Remember, we will be injecting this service into the pages on our application so that we can control operations related to the addition, updating, or deletion of checklists and their items.

Let's start off by setting up a bit of a "skeleton" version of our service. After this, we will work through adding each of the functions in more detail.

> Modify `src/app/services/checklist.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { Storage } from "@ionic/storage";
import { Observable, BehaviorSubject } from "rxjs";
import { map } from "rxjs/operators";
import { Checklist } from "../interfaces/checklist";

@Injectable({
  providedIn: "root",
})
export class ChecklistService {
  private checklists$: BehaviorSubject<Checklist[]> = new
BehaviorSubject<Checklist[]>([]);
  private checklists: Checklist[] = [];
  private loaded: boolean = false;

  constructor(private storage: Storage) {}

  async load(): Promise<void> {
    return Promise.resolve();
  }

  getChecklists(): Observable<Checklist[]> {}
```

```
  getChecklist(checklistId: string): Observable<Checklist> {}

  async createChecklist(title: string): Promise<void> {}

  async updateChecklist(checklistId: string, newTitle: string):
Promise<void> {}

  async removeChecklist(checklistId: string): Promise<void> {}

  async addItemToChecklist(checklistId: string, title: string):
Promise<void> {}

  removeItemFromChecklist(checklistId: string, itemId: string):
void {}

  updateItemInChecklist(checklistId: string, itemId: string,
newTitle: string): void {}

  setItemStatus(checklistId: string, itemId: string, checked:
boolean): void {}

  resetItemStatusForChecklist(checklistId: string): void {}

  save(): Promise<void> {
    return Promise.resolve();
  }

  generateSlug(title: string): string {
    return "";
  }
```

```
}
```

We have a basic service/provider/injectable set up here with a bunch of empty functions, and a class member of `checklists`. Notice that we have imported our interface and we are using this as the type for our class member:

```
private checklists: Checklist[] = [];
```

This variable will be an array of checklists. This array will contain all of the checklists that are currently available in the application.

Each of the functions in this service will allow us to perform some kind of operation on our checklists or their items, and we will be stepping through implementing those one-by-one. Since we will not be completely implementing the `load` and `save` methods just yet, we are just returning dummy promises so that we don't get errors because of the return type on these methods of `Promise<void>` (which indicates that the method should return a promise).

Important concepts to understand before we begin

We will be using modern development concepts and best practices to implement this service, and when all of this is put together it can look quite complex. There are a few key concepts we will be making use of, so it is worth having this in mind beforehand so that the code itself doesn't become too overwhelming. If you do not feel comfortable with any

of these key concepts, it might be worth brushing up on them a little first.

Mutability and Immutability

Mutability and **Immutability** are popular terms you will hear in modern web development, which essentially translates to:

- **Mutable:** can be changed
- **Immutable:** can **not** be changed

Our service will maintain a single `checklists` data source that is an array of all of the checklists and their items. We are going to treat this array as being **immutable** which means that we will not change it. But how are we to add new checklists or edit a title if we can't change the data? This is a common pattern used in modern web development - instead of modifying the existing array, we will instead return a *new* array that contains the data changes we want.

For example, if we were treating our array as being **mutable** we might do something like this to add a new checklist:

```
this.checklists.push(newChecklist);
```

We have modified the existing array to make this change. However, if we were treating our as being **immutable** we could make the same change like this:

```
this.checklists = [...this.checklists, newChecklist];
```

This time, we have created an entirely new array. We used the `spread operator` to fill the new array with all of the values from our existing array, and then added our `newChecklist` on to the end.

The `spread operator`, which is the `...` that you see above, is important to understand well when using this approach. The spread operator will allow us to copy over the values from an existing array/object into a new one, without needing to modify the existing array/object. In an overly simplified sense, the spread operator will "unpack" all of the elements/properties of an array/object. Let's say we have an array like this:

```
const myArray = [1, 2, 3];
```

If we wanted to pull all of these values into a new array, we might try to do something like this:

```
const myBiggerArray = [myArray, 4, 5];
```

This doesn't work, because the end result will be:

```
[[1, 2, 3], 4, 5]
```

However, if we do this:

```
const myBiggerArray = [...myArray, 4, 5];
```

The result will be:

```
[1, 2, 3, 4, 5]
```

The original `myArray` will remain intact and unmodified, but we will have "added" new elements to `myArray` by creating an entirely new array with the values of `myArray` plus some extra ones.

This is just a basic example, so if this is new to you, I would highly recommend reading up on this concept first:

- Spread syntax

Why does this matter? You might be wondering why we should fuss over this when we could just as well just modify the original array and forget all this nonsense about mutability and immutability. In our case, it doesn't strictly matter whether we modify the existing array

or not, but it is a good pattern to follow in general that can reduce bugs/unexpected behaviour. Once you get used to maps and filters, they are also a very productive way to go about modifying arrays.

Observables/BehaviorSubject

We have already discussed what an `Observable` is in the basics section of this book. To recap, we can `subscribe` to an observable and set up some logic to handle data that the observable emits over time. You might have noticed that we have set up an additional `checklists$` member variable as well as our `checklists` member variable that looks like this:

```
private checklists$: BehaviorSubject<Checklist[]> = new
BehaviorSubject<Checklist[]>([]);
private checklists: Checklist[] = [];
```

The `$` syntax does not do anything special here, it is just a common way to denote that a particular variable is an observable - the `$` doesn't actually *do* anything and you don't have to use it if you don't want to. However, it is useful here to distinguish between our `checklists` array that we will use internally in the checklists service to manage the data, and the data that we will be exposing to other parts of our applications through this `checklists$` observable.

But what is a BehaviorSubject?

A `BehaviorSubject` is a type of observable. I say a *type* of observable because it is a little different to a standard observable. We subscribe to a `BehaviourSubject` just like we would a normal observable, but the benefit of a `BehaviourSubject` for our purposes is that:

- It will always return a value, even if no data has been emitted from its stream yet
- When you subscribe to it, it will immediately return the last value that was emitted immediately (or the initial value if no data has been emitted yet)

We have set our `BehaviorSubject` up to emit our `checklists` array, which is why we use this syntax:

```
BehaviorSubject<Checklist[]>
```

This indicates that `checklists$` is a `BehaviorSubject` that will emit `Checklist[]` values. We also initialise the `BehaviorSubject` with an empty array:

```
new BehaviorSubject<Checklist[]>([]);
```

This means that if we `subscribe` to `checklists$` before the data has finished loading into the application we will just get a empty array (which is what we want). When the data is loaded into the application, our subscription to `checklists$` will notify whatever is subscribed to it of the new data. If we subscribe to `checklists$` *after* the data has

already loaded then that is fine too, because the `BehaviorSubject` will just return us the most recent data that it has been updated with.

This behavior is perfect for us. It means that from anywhere in our application we can subscribe to `checklists$` and be confident that we are receiving the latest data available.

Map and Filter

We will be making heavy use of the `map` and `filter` methods of arrays in this lesson. We can use the `map` method of an array when we want to *change* the values in the array somehow, and we can use the `filter` method of an array when we want to remove some elements in that array. If you are not already comfortable with using `map` and `filter` I would recommend reviewing those before continuing:

- map
- filter

Now let's start implementing our services methods one by one.

load & save

The `load` function will be responsible for initialising the `checklists` member variable with data loaded in from storage. Similarly, the `save` function will be responsible for saving data to storage. However, we will be implementing this in another lesson as we will be talking about data storage in more depth. For now, all data will be lost when we refresh the

application.

getChecklists

This method will return us an observable that emits all of the data for our checklists.

> Modify the `getChecklists` method to reflect the following:

```
getChecklists(): Observable<Checklist[]> {
  return this.checklists$;
}
```

This method does not achieve a lot. Technically, we could just make the `checklists$` observable public and subscribe directly to that, but having a method that returns the observable for us is better practice. This way, if we were to change the internals of our service (e.g. by renaming the `checklists$` observable) we don't need to worry about how that might be affecting the rest of our application.

getChecklist

This method has the same idea as the previous one, except that in this case we want to modify the value returned by the observable (all of the checklists) so that it only includes the **one** checklist we are interested in (the one that matches the `checklistId` supplied).

> Modify the `getChecklist` method to reflect the following:

```
getChecklist(checklistId: string): Observable<Checklist> {
  return this.checklists$.pipe(
    map((checklists) => checklists.find((checklist) =>
checklist.id === checklistId))
  );
}
```

The `map` being used here should not be confused with the `map` method of an array. Since we want to get rid of all of the checklists except the one with the matching `checklistId` you might initially think that a `filter` would be the appropriate approach here. However, keep in mind that the when we are using the `map` operator in the context of an observable, it is mapping whatever value the observable emits. In our case, the value being emitted is just one value, and that value is an array of all of our checklists. We can't `filter` this because it is just one value. What we need to do is modify that one value so that it only includes the value we are interested in. To do this, we `map` the `checklists` and return the result of `checklists.find()` where we use the `checklistId` to find the checklist with the matching `id`.

createChecklist

This function will be responsible for the creation of a new checklist.

> Modify the `createChecklist` method to reflect the following:

```
async createChecklist(title: string): Promise<void> {
  await this.load();

  const newChecklist = {
    id: this.generateSlug(title),
    title: title,
    items: [],
  };

  this.checklists = [...this.checklists, newChecklist];
  await this.save();
}
```

Before we try to create a new checklist, we make sure that the existing checklist data has been loaded by awaiting the `load()` method. We then create our `newChecklist` object with the supplied title, and then we use that to create a **new** array containing all of the existing checklists plus our one new one.

This is where those immutability concepts come into play. We create a new array, and use the spread operator to fill that array with all of the existing checklists:

```
this.checklists = [...this.checklists];
```

At this point, we are just overwriting our old array with a new array that contains all of the same values... except we also add our new checklist onto the end:

```
this.checklists = [...this.checklists, newChecklist];
```

Now we have a new array that contains all of the old checklists, plus our new one, and we overwrite the old array with the new array on our `this.checklists` member variable.

Notice that we call the `generateSlug` method to generate the `id`. We will implement this function in a moment, and its responsibility is basically just to turn the name of the checklist into a unique `id`. Using a sensical `id` like `my-checklist` rather than an `id` like `0938242` makes for more human-readable URLs.

Finally, we call `this.save()` to trigger the saving of the data and to update the `checklists$` observable. Remember that we won't be implementing the `save` functionality fully until later.

updateChecklist

Next, we're going to define the `updateChecklist` function which will allow us to supply a new title for an existing checklist.

> Modify the `updateChecklist` method to reflect the following:

```
async updateChecklist(checklistId: string, newTitle: string):
Promise<void> {
  this.checklists = this.checklists.map((checklist) => {
    return checklist.id === checklistId ? { ...checklist, title:
newTitle } : checklist;
  });

  this.save();
}
```

We are taking the same approach here as we will with just about every other method in this service - overwrite the existing `this.checklists` array with a new one. In this case, we want to modify the values in the new array such that the checklist with an `id` of `checklistId` has its `title` changed to `newTitle`.

To do this, we `map` all the elements in `this.checklists` and perform this check:

```
return checklist.id === checklistId ? { ...checklist, title:
newTitle } : checklist;
```

This uses the ternary operator (which is basically just shorthand for `if/else`) to check if the `id` matches. If it does not match, then we just return the `checklist` with no modification. If it does match then we modify the element by returning this:

```
{
  ...checklist,
  title: newTitle
}
```

Now we are using the spread operator with an object. What this will do:

```
{
  ...checklist
}
```

...is create a new object will all of the same properties/values as `checklist`. However, when we do this:

```
{
  ...checklist,
  title: newTitle
}
```

...it will still create a new object with all of the same properties/values as checklist, but it will overwrite its `title` property with our `newTitle` value.

removeChecklist

Next up we are going to add the ability to delete a checklist.

> Modify the `removeChecklist` method to reflect the following:

```
async removeChecklist(checklistId: string): Promise<void> {
  this.checklists = this.checklists.filter((checklist) =>
checklist.id !== checklistId);
  this.save();
}
```

This time, we just need to filter out any elements that have an `id` of `checklistId`. If the `id` does not match then it will pass through the filter and be kept, if it does match then it will be filtered out.

addItemToChecklist

We have added plenty of functions for modifying the data of our checklists, now we will need some to modify the items within a checklist. This one will add an item to a specific checklist (i.e. an item within that checklist that can be marked as complete/incomplete).

> Modify the `addItemToChecklist` method to reflect the following:

```
async addItemToChecklist(checklistId: string, title: string):
Promise<void> {
  const newItem = {
    id: Date.now().toString(),
    title: title,
    checked: false,
  };

  this.checklists = this.checklists.map((checklist) => {
    return checklist.id === checklistId
      ? { ...checklist, items: [...checklist.items, newItem] }
      : checklist;
  });

  this.save();
}
```

First, we create our `newItem`. It needs a unique `id`, but unlike our checklist `id` that we wanted to use in the URL (so we wanted it to be readable), this `id` does not need to look nice so we just use `Date.now().toString()` to get a unique value.

We are still ultimately using the same concepts before with mapping, filtering, and the spread operator, but it gets a bit harder to follow now because we are dealing with nested data (an array of items inside of an object which is itself also part of an array inside of an object).

In this case, we want to modify one of the elements inside of our `checklists` array so we use a map. When we find the checklist with the `id` that we are interested in we return the following:

```
{
  ...checklist,
  items: [...checklist.items, newItem]
}
```

This will return a new object with all of the same properties/values as the original `checklist`, but it will overwrite the `items` property with:

```
[...checklist.items, newItem]
```

This also uses the spread operator to take all of the original items contained within `checklist.items`, but then adds our `newItem` to that.

removeItemFromChecklist

This method will remove an item from within an individual checklist.

> Modify the `removeItemFromChecklist` method to reflect the following:

```
removeItemFromChecklist(checklistId: string, itemId: string): void
{
  this.checklists = this.checklists.map((checklist) => {
    return checklist.id === checklistId
      ? { ...checklist, items: [...checklist.items.filter((item)
=> item.id !== itemId)] }
      : checklist;
  });
  this.save();
}
```

This uses more or less the same concept as the previous method, except instead of adding a new item into the `items` array we are filtering one out:

```
[...checklist.items.filter((item) => item.id !== itemId)]
```

updateItemInChecklist

This method will be used to update the title of a specific item within a checklist.

> Modify the `updateItemInChecklist` method to reflect the following:

```
updateItemInChecklist(checklistId: string, itemId: string,
newTitle: string): void {
  this.checklists = this.checklists.map((checklist) => {
    return checklist.id === checklistId
      ? {
          ...checklist,
          items: [
            ...checklist.items.map((item) =>
              item.id === itemId ? { ...item, title: newTitle } :
item
            ),
          ],
        }
      : checklist;
  });
  this.save();
}
```

We are getting into kind of tricky territory now, but if you break it down it isn't so bad. Still, the same initial concept applies as in the previous two methods - we map our `this.checklists` array to find the checklist we are interested in, and then when we find that one we modify its value to:

```
{
  ...checklist,
```

```
  items: [
    ...checklist.items.map((item) =>
      item.id === itemId ? { ...item, title: newTitle } : item
    ),
  ],
}
```

Again, we use the spread operator to create a new object with all of the properties/values of the initial `checklist` except we overwrite the `items` array with:

```
[
  ...checklist.items.map((item) =>
    item.id === itemId ? { ...item, title: newTitle } : item
  )
]
```

Instead of just adding all of the existing `checklist.items` immediately to the new array, it first maps them. This `map` operation will find the `item` we are interested in by its `id` and when it finds it, it will return this value instead of its normal value:

```
{
  ...item,
  title: newTitle
```

```
}
```

This will create a new object with all of the existing properties/values of the original `item` except it will overwrite its `title` with the `newTitle` value.

setItemStatus

Each of the items in our checklist can be toggled between a completed and incomplete state. This function flips the value of a particular item and then saves it to storage.

> Modify the `setItemStatus` method to reflect the following:

```
setItemStatus(checklistId: string, itemId: string, checked:
boolean): void {
  this.checklists = this.checklists.map((checklist) => {
    return checklist.id === checklistId
      ? {
          ...checklist,
          items: [
            ...checklist.items.map((item) =>
              item.id === itemId ? { ...item, checked: checked } :
item
            ),
          ],
        }
```

```
      : checklist;
  });
  this.save();
}
```

This is exactly the same as the `updateItemInChecklist` method, we are just overwriting the `checked` property in the end instead of the `title` property.

resetItemStatusForChecklist

As well as being able to change the `checked` status for one individual item, we also want to provide a way to set the `checked` status for *all* of an individual checklist's items back to `false` (i.e. for when the user has completed the checklist and wants to restart from the beginning again).

> Modify the `resetItemStatusForChecklist` method to reflect the following:

```
resetItemStatusForChecklist(checklistId: string): void {
  this.checklists = this.checklists.map((checklist) => {
    return checklist.id == checklistId
      ? {
          ...checklist,
          items: [
            ...checklist.items.map((item) => {
              return { ...item, checked: false };
```

```
            }),
          ],
        }
      : checklist;
  });

  this.save();
}
```

Again, more or less the exact same concept, except we don't need to find a specific item, and we just use a `false` value instead of a passed in value.

save

I've mentioned that we won't be implementing the `save()` method in full just yet, since we are dealing with data storage later, but we do still need to emit a value on our `checklists$` observable if we want the rest of our application to have access to the new data.

> Modify the save method to reflect the following:

```
save(): Promise<void> {
  this.checklists$.next(this.checklists);
  return Promise.resolve();
}
```

By calling the `next` method and supplying it with the latest `this.checklists` value, anywhere in the application that we have a subscription set up to `checklists$` will automatically receive the data and can do whatever is necessary with that data.

generateSlug

Finally, we come to perhaps our most interesting function. Let's take a look.

> Modify the `generateSlug` method to reflect the following:

```
generateSlug(title: string): string {
  // NOTE: This is a simplistic slug generator and will not handle
things like special characters.
  let slug = title.toLowerCase().replace(/\s+/g, "-");

  // Check if the slug already exists
  const matchingSlugs = this.checklists.filter((checklist) => {
    return checklist.id.substring(0, slug.length) === slug;
  });

  // If the title is already being used, add a string to make the
slug unique
  if (matchingSlugs.length > 0) {
    slug = slug + Date.now().toString();
  }
```

```
  return slug;
}
```

As I mentioned, we need a unique `id` for each of our checklists, but we want the ids to look nice and make sense in the URL. To do this, we will be using a hyphenated version of the title for the checklist. However, if a user were to create two checklists with the same title then it would no longer be unique. This function checks if the `id` already exists, and if it does it will add a unique string to the end of the `id` slug.

We are using a very simple regex pattern to replace the spaces in the checklists title with hyphens, in a "serious" application you might want to modify this to be more robust.

We have finished implementing our checklist service - now, you will be able to import this service into any other page/component/service you like, inject it into the constructor, and then use it as you please!

You could check your application in the browser with `ionic serve` now if you like, but nothing will have changed as a result of this lesson. We have created this service to be utilised throughout the application, but we haven't actually made use of it anywhere yet.

Lesson 5: Creating Checklists and Checklist Items

We have our template that allows the user to interact with the application, and we have our checklist service that handles data operations, now we just need to get those two elements of our application talking to each other.

In this lesson, we will be making modifications to both our **home page** that displays all available checklists, and our **checklist page** that displays items for a particular checklist. These modifications will allow us to interact with various features like adding and deleting checklists. We already have most of the logic for this built into our service, we just need to make the appropriate calls to the service.

Implementing the Home Page

Although we have the basic template for our home page set up, it doesn't do much. We are making calls to functions that don't exist, and we are just using dummy data in place of actual data.

In order to get everything working as it should, we are first going to implement the required logic in our **home.page.ts** file, and then we will update the template as necessary. Generally, the main purpose of the logic that we add to our page components is to set up simple bindings for the functions we are calling in the template. Clicking on a button in the template will trigger some function in the logic for that page, and then that function can make calls to additional services if necessary. Remember, the logic that we add to our pages should be very simple and mostly relate directly to displaying the page and responding to user input, any complex logic should be handled through services like our

checklist service.

Just like last time, we are going to create a basic outline for our page, and then we will step through implementing each of the functions.

> Modify `src/app/home/home.page.ts` to reflect the following

```
import { Component, ViewChild, OnInit } from "@angular/core";
import { AlertController, IonList, IonContent, NavController }
from "@ionic/angular";
import { ChecklistService } from "../services/checklist.service";
import { Checklist } from "../interfaces/checklist";

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage implements OnInit {
  @ViewChild(IonList, { static: false }) slidingList: IonList;
  @ViewChild(IonContent, { static: false }) contentArea:
IonContent;

  public checklists: Checklist[] = [];

  constructor(
    private checklistService: ChecklistService,
```

```
    private alertCtrl: AlertController,
    private navCtrl: NavController
  ) {}

  ngOnInit() {}

  async addChecklist(): Promise<void> { }

  async renameChecklist(checklist: Checklist): Promise<void> { }

  async removeChecklist(checklist: Checklist): Promise<void> { }
}
```

One thing you might notice here is that we are importing something called `AlertController`. This is provided by the `@ionic/angular` package and will allow us to create alert components to display to the user. You could use an "alert" for a variety of purposes, but in this instance, we are going to use it to allow the user to enter a name for a new checklist (or supply a new name for an existing checklist). We inject this `AlertController` along with our `ChecklistService` into the constructor to make it available throughout the class.

We are also importing `IonList`, `IonContent`, and `ViewChild`. We can use `@ViewChild` as we do above the `constructor` to grab a reference to an element in our template. By importing `IonList` and `IonContent` we can easily tell `ViewChild` to look for instances of these components in our template. In this case, we are telling it to look for an `IonList`, and assign it to the member variable `slidingList`. This will be a reference to the `<ion-list>` in our template. The reason we want a reference to this is so that we

can call the `closeSlidingItems()` method on the list, which will automatically close any of the sliding items we have open in the list.

We also set up a reference to our `<ion-content>` and assign it to `contentArea`. The reason we do this is because we will want to make use of the `scrollToBottom()` method that the `<ion-content>` component provides. We will utilise this to scroll the list to the bottom when a new checklist is added. This means that if a user had `30` checklists in their list and they were currently at the top of the list when they added another new checklist, the list would automatically scroll down so that they can see the new checklist that they just added.

We have a `public` member variable set up for our `checklists` which means we will be able to access it from our template. Everything that we have injected through the `constructor` is `private` which means we will only be able to access them from within this class (not the template).

Let's start implementing the methods now.

ngOnInit

Our home page implements `OnInit` which allows us to set up an `ngOnInit` lifecycle hook that will run when the home page is initialised. We are going to use this to subscribe to the `checklists$` behavior subject/observable from our checklists service.

> Modify the `ngOnInit` method to reflect the following:

```
  ngOnInit() {
    this.checklistService.getChecklists().subscribe((checklists)
=> {
      this.checklists = checklists;
    });
  }
```

Since we are subscribing the `getChecklists()` which returns out `checklists$` observable, this line:

```
this.checklists = checklists;
```

...will run every time new checklist data is received. This means that as soon as any data us updated in our service, our home page will update to reflect that change.

addChecklist

> Modify the `addChecklist` method to reflect the following:

```
  async addChecklist(): Promise<void> {
    const alert = await this.alertCtrl.create({
      header: "New Checklist",
      message: "Enter the name of your new checklist below:",
```

```
      inputs: [
        {
          type: "text",
          name: "name",
        },
      ],
      buttons: [
        {
          text: "Cancel",
        },
        {
          text: "Save",
          handler: async (data) => {
            await
this.checklistService.createChecklist(data.name);
            this.contentArea.scrollToBottom(300);
          },
        },
      ],
    });

    alert.present();
  }
```

The goal of this function is to display an alert to the user that enabled them to enter some text, and then send the data they enter off to our checklist service.

We call the `create` method of the alert controller to create a new alert. There are many ways we could configure this, but we are just supplying some `header` text, a `message`, a `text` input, and we also add two buttons to the alert. One button will just allow the user to cancel adding a new checklist, but the other will handle sending the data off to the checklist service. Also notice that we `await` the promise that is returned by calling `createChecklist` and once that resolves we use the `scrollToBottom` method to scroll the content area to the bottom (which is where our new checklist will be).

The `create` method will return a promise that resolves once the alert has been created. This is why we `await` the `create()` method and assign the result to the `alert` constant. Once the promise has resolved, we call the `present` method to display it to the user.

renameChecklist

> Modify the `renameChecklist` method to reflect the following:

```
async renameChecklist(checklist: Checklist): Promise<void> {
  const alert = await this.alertCtrl.create({
    header: "Rename Checklist",
    message: "Enter the new name of this checklist below:",
    inputs: [
      {
        type: "text",
        name: "name",
      },
    ],
```

```
      buttons: [
        {
          text: "Cancel",
        },
        {
          text: "Save",
          handler: async (data) => {
            await
this.checklistService.updateChecklist(checklist.id, data.name);
            this.slidingList.closeSlidingItems();
          },
        },
      ],
    });

    alert.present();
  }
```

This is basically the exact same idea, except we are using this to rename a checklist instead, and so we send the appropriate data off to the `updateChecklist` method in our checklist service. Since the user will have opened a sliding item to click the edit button, we make sure to close the sliding items once the edit has been finished (otherwise, the user would need to manually swipe the item closed again which would be annoying).

removeChecklist

> Modify the `removeChecklist` method to reflect the following:

```
  async removeChecklist(checklist: Checklist): Promise<void> {
    await this.slidingList.closeSlidingItems();
    this.checklistService.removeChecklist(checklist.id);
  }
```

All this function does is pass the checklist detail on to the `removeChecklist` function in the checklist service, as no additional data is needed from the user in order to delete a checklist.

With all of the logic for our functions defined, we now just need to make a few updates to the template.

> Modify `src/app/home/home.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="success">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="addChecklist()">
        <ion-icon slot="icon-only" name="add"></ion-icon>
```

```
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <ion-header collapse="condense">
    <ion-toolbar>
      <ion-title color="light" size="large">Your lists</ion-title>
    </ion-toolbar>
  </ion-header>

  <div id="container">
    <ion-list lines="none">
      <ion-item-sliding *ngFor="let checklist of checklists">
        <ion-item routerLink="/checklists/{{checklist.id}}"
routerDirection="forward">
          <ion-label>{{ checklist.title }}</ion-label>
          <span>{{ checklist.items.length }} items</span>
        </ion-item>

        <ion-item-options side="end">
          <ion-item-option color="light"
(click)="renameChecklist(checklist)">
            <ion-icon slot="icon-only" name="clipboard"></ion-
icon>
          </ion-item-option>
          <ion-item-option color="danger"
```

```
(click)="removeChecklist(checklist)">
            <ion-icon slot="icon-only" name="trash"></ion-icon>
          </ion-item-option>
        </ion-item-options>
      </ion-item-sliding>
    </ion-list>
  </div>
</ion-content>
```

This is mostly the same as it was before, but with a few important differences. Of course, all of the functions that we were calling before now actually link up to something in our class definition. Most importantly, we have modified this:

```
      <ion-item-sliding *ngFor="let checklist of checklists">
        <ion-item routerLink="/checklists/{{checklist.id}}"
routerDirection="forward">
          <ion-label>{{ checklist.title }}</ion-label>
          <span>{{ checklist.items.length }} items</span>
        </ion-item>

        <!-- ion-item-options here -->

      </ion-item-sliding>
```

We are now using the structural directive `*ngFor` to create an `<ion-item-sliding>` for

every checklist that is in the `checklists` array (the `this.checklists` member variable in **home.page.ts**). Inside of this loop, we are then able to access details about each specific checklist. Now we are able to use interpolations like this:

```
{{ checklist.title }}
```

to grab the `title` of the specific checklist that is being looped over at the time. We are also utilising data from one of the checklists inside of the `routerLink` for the `ion-item`. Notice that we are also using an interpolation of `{{checklist.id}}` in the `routerLink` because we want the route to be `/checklists/` plus the `checklist.id` for the specific checklist. By doing this we are now able to supply the checklist page with the data it needs to fetch and display a specific checklist.

We also use interpolations to display the title and number of items in each checklist.

Checklist Page

As I just mentioned, we now have a way to pass in details about a specific checklist to our checklist page. Now we need to implement the functionality on this page to fetch and display that checklist, and we also need to create bindings from our template to our class for the other features this page makes available (like marking items as complete, adding new items, and so on).

Once again, let's start with a skeleton and then work from there.

> Modify `src/app/checklist/checklist.page.ts` to reflect the following

```
import { Component, ViewChild, OnInit, OnDestroy } from
"@angular/core";
import { ActivatedRoute } from "@angular/router";
import { AlertController, IonList } from "@ionic/angular";
import { ChecklistService } from "../services/checklist.service";
import { Checklist } from "../interfaces/checklist";
import { ChecklistItem } from "../interfaces/checklist-item";
import { Subscription } from "rxjs";

@Component({
  selector: "app-checklist",
  templateUrl: "./checklist.page.html",
  styleUrls: ["./checklist.page.scss"],
})
export class ChecklistPage implements OnInit, OnDestroy {
  @ViewChild(IonList, { static: false }) slidingList: IonList;
  private checklistSubscription: Subscription;

  public checklist: Checklist = {
    id: "",
    title: "",
    items: [],
  };

  constructor(
```

```
    private alertCtrl: AlertController,
    private route: ActivatedRoute,
    private checklistService: ChecklistService
  ) {}

  ngOnInit() {}

  ngOnDestroy() {}

  async addItem(): Promise<void> {}

  async renameItem(item: ChecklistItem): Promise<void> {}

  async removeItem(item: ChecklistItem): Promise<void> {}

  toggleItem(item: ChecklistItem): void {}

  restartList(): void {}

}
```

Some of these methods will look very similar to the methods on the home page, but we also have some more interesting stuff happening here as well. First of all, you might notice that we are now importing and injecting `ActivatedRoute` - this is the first time we have seen this. Since we are passing in an `id` through the URL, using `ActivatedRoute` will allow us to grab the data that is being passed in and use it. We also have two lifecycle hooks now: `ngOnInit` and `ngOnDestroy` - you will see why in just a moment. Another thing to note is how we are initialising our `checklist` member variable with a

dummy/blank checklist. This is because our template will be referencing values in this `checklist` but initially it won't be defined because we first need to grab the `id` that is being passed in and retrieve the matching checklist from our checklists service. By setting a blank `checklist` initially, our template can still reference values like `checklist.title` and not receive an error - it will just use a blank title until the `checklist` is updated with the proper value.

Let's get into defining the methods now.

ngOnInit

The `ngOnInit` lifecycle hook will run when the component is initialised, and we are going to use this to load up the specific checklist that we are trying to view.

> Modify the `ngOnInit` method to reflect the following:

```
  ngOnInit() {
    const id = this.route.snapshot.paramMap.get("id");
    this.checklistSubscription =
this.checklistService.getChecklist(id).subscribe((checklist) => {
      this.checklist = checklist;
    });
  }
```

We use this method to grab the `id` of the particular checklist that we are trying to view. We

use the `ActivatedRoute` that we injected into this class to grab the `id` value from the `paramMap`. We then pass that `id` into our `getChecklist` method from our checklists service, which will return us our `checklists$` observable with all of the checklists except the one we are interested in filtered out. Again, since we have a subscription to our observable set up, this code:

```
this.checklist = checklist;
```

...will update the checklist data any time that it is changed in our checklists service.

ngOnDestroy

The `ngOnDestroy` lifecycle hook will run when the component is destroyed (e.g. when we navigate back to our home page). We are going to use this lifecycle hook to **unsubscribe** from the observable returned by `getChecklist()`.

> Modify the `ngOnDestroy` method to reflect the following:

```
ngOnDestroy() {
  if (this.checklistSubscription !== null) {
    this.checklistSubscription.unsubscribe();
  }
}
```

This isn't strictly necessary, but if we do not `unsubscribe` from subscriptions we set up it can lead to memory leaks and other issues. Consider that the user might have `30` different checklists. Each time they go into a checklist we will `subscribe` to `getChecklist(id)` and it will continue emitting data every time a change occurs in the checklists service. If we do not `unsubscribe` when we leave the page (at which point we no longer need the subscription anyway) then these subscriptions will just keep building up over time.

This isn't an issue for the subscription on our home page, because the home page is only initialised once (so we will only have a single subscription) and we do want to keep receiving data updates on the home page.

addItem, removeItem, and renameItem

I have bunched these methods together because it is exactly the same idea that we used on our home page for `addChecklist`, `removeChecklist`, and `renameChecklist`.

> Modify the `addItem`, `removeItem`, and `renameItem` methods to reflect the following:

```
  async addItem(): Promise<void> {
    const alert = await this.alertCtrl.create({
      header: "Add Item",
      message: "Enter the name of the task for this checklist
below:",
      inputs: [
        {
```

```
          type: "text",
          name: "name",
        },
      ],
      buttons: [
        {
          text: "Cancel",
        },
        {
          text: "Save",
          handler: (data) => {

this.checklistService.addItemToChecklist(this.checklist.id,
data.name);
          },
        },
      ],
    });

    alert.present();
  }

  async renameItem(item: ChecklistItem): Promise<void> {
    const alert = await this.alertCtrl.create({
      header: "Rename Item",
      message: "Enter the new name of the task for this checklist
below:",
      inputs: [
```

```
        {
          type: "text",
          name: "name",
        },
      ],
      buttons: [
        {
          text: "Cancel",
        },
        {
          text: "Save",
          handler: (data) => {

this.checklistService.updateItemInChecklist(this.checklist.id,
item.id, data.name);
            this.slidingList.closeSlidingItems();
          },
        },
      ],
    });

    alert.present();
  }

  async removeItem(item: ChecklistItem): Promise<void> {
    await this.slidingList.closeSlidingItems();

this.checklistService.removeItemFromChecklist(this.checklist.id,
item.id);
```

```
  }
```

toggleItem

> Modify the `toggleItem` method to reflect the following:

```
  toggleItem(item: ChecklistItem): void {
    this.checklistService.setItemStatus(this.checklist.id,
item.id, !item.checked);
  }
```

This function just has the responsibility of calling the `setItemStatus` method in the checklist service.

restartList

> Modify the `restartList` method to reflect the following:

```
  restartList(): void {
this.checklistService.resetItemStatusForChecklist(this.checklist.id
  }
```

Instead of just toggling the completed state of one item, this will use our checklists service to reset the completed state for all of the items in this checklist.

Now that we have all of our functions implemented, we will also need to make some changes to the template for this page.

> Modify `src/app/checklist/checklist.page.html` to reflect the following

```
<ion-header class="ion-no-border">
  <ion-toolbar color="success">
    <ion-title> {{ checklist.title }} </ion-title>
    <ion-buttons slot="start">
      <ion-back-button color="light" defaultHref="/checklists">
</ion-back-button>
    </ion-buttons>
    <ion-buttons slot="end">
      <ion-button (click)="restartList()">
        <ion-icon slot="icon-only" name="refresh"></ion-icon>
      </ion-button>
      <ion-button (click)="addItem()">
        <ion-icon slot="icon-only" name="add"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

```
<ion-content>
  <ion-list lines="none">
    <ion-item-sliding *ngFor="let item of checklist.items">
      <ion-item color="success">
        <ion-label>{{ item.title }}</ion-label>
        <ion-checkbox
          color="light"
          [checked]="item.checked"
          (ionChange)="toggleItem(item)"
        ></ion-checkbox>
      </ion-item>

      <ion-item-options side="end">
        <ion-item-option color="light" (click)="renameItem(item)">
          <ion-icon slot="icon-only" name="clipboard"></ion-icon>
        </ion-item-option>
        <ion-item-option color="danger"
(click)="removeItem(item)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

We have made pretty much the same changes to this template as we did with our home

page - we are just using an `*ngFor` loop and some interpolations.

You should now be able to perform just about every function of the application, which includes creating checklists, modifying them, viewing individual checklists, and adding items to individual checklists.

Try running the application in your browser by running `ionic serve` and adding your own checklists and checklist items. We do still need to make the application look a little prettier, and we need to handle saving and loading data (so that checklists aren't lost when we refresh the application). We will be taking care of these in the next lessons.

IMPORTANT: If you are viewing the application with the **iOS** styling, then you won't be able to see the "Add" button due to the current styling (which we will obviously be changing). On iOS, there is no toolbar visible initially, so the white add button gets lost on the white background. If you click in the top-right corner, you will be able to test the functionality of the application still.

NOTE: Remember, if you are running into any issues (e.g. errors in the console, functionality not working, or compilation errors in the terminal) make sure to stop serving (`CTRL + C`) and re-serve the application with `ionic serve`. Sometimes the live reloading fails to update the application properly, and you will need to re-serve the application.

Lesson 6: Saving and Loading Data

You know what would be really annoying? If you create an entire checklist full of items for some task you need to complete, come back to use it later and it's just gone. Well, that's exactly how the application works right now, so we are going to need to add a way to save any data the user adds to the application for use later.

We've already set up a lot of the structure for this, we already call the `save` method every time some data changes, we just need to utilise that method to actually save the data to storage. We will also need to handle triggering the loading of our data.

To do this, we will be making use of Ionic's own storage service. We covered what this does in the basics section, but let's quickly recap.

Storage is Ionic's generic storage service, and it handles storing data in the best way possible whilst providing a consistent API for us to use.

When running on a device, and if the SQLite plugin is available (which we installed earlier), it will store data using a native SQLite database. Since the SQLite database will only be available when running natively on a device, `Storage` will also use `IndexedDB`, `WebSQL`, or standard browser `localStorage` if the SQLite database is not available.

It's best to use SQLite where possible because the browser-based local storage is not completely reliable and can potentially be wiped by the operating system. Having your data wiped randomly is obviously not ideal.

Remember that in order to use the storage service, you need to make sure you add the `IonicStorageModule` to the root module in **app.module.ts**. We have already set this up in the **Getting Ready** lesson.

Modifying the Checklist Service

First, we will modify our checklist service so that it can save and load data.

> Modify the `save` method `src/app/services/checklist.service.ts` to reflect the following:

```
save(): Promise<void> {
  this.checklists$.next(this.checklists);
  return this.storage.set("checklists", this.checklists);
}
```

The Storage API uses a simple key-value system for storage. All we are doing here is calling the `set` method and supplying it with a `key` of `checklists` and then we supply it with our array of checklists (which is the data that we want to save). Later, we will be able to retrieve this same data by referencing the `checklists` key. Let's do that now.

> Modify the `load` method `src/app/services/checklist.service.ts` to reflect the following:

```
async load(): Promise<void> {
  if (!this.loaded) {
    const checklists = await this.storage.get("checklists");
    if (checklists !== null) {
      this.checklists = checklists;
      this.checklists$.next(this.checklists);
    }

    this.loaded = true;
  }
}
```

This time, we call the `get` method of the storage service, and we supply it with the `key` we are trying to retrieve. This will return a promise, so we use `await` to retrieve this value. Once we do get the data, we check to see if it is not empty (i.e. nothing has been saved yet), and then we update our `this.checklists` member variable with the data. We also trigger the `next` method on our `checklists$` behavior subject so that anything that is subscribed to it will receive this data that has been loaded in.

This method is set up in such a way that it won't matter if `load()` is called from multiple places. Since we set the `loaded` flag after it has loaded, the data will only ever be loaded in once. If `load()` is called after the data has already been loaded, the promise that `load()` returns will just resolve instantly.

Now we can call this `load` function from anywhere in the application to initialise our

`checklists` array in the checklists service.

Loading Data

We have a way to load the data into our application now, but we still need to call that `load` function from somewhere. We are going to do this in our root component. Our root component is the first component added to the application, so that means our data can be loaded right away.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { ChecklistService } from "./services/checklist.service";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(private checklistService: ChecklistService) {
    this.initializeApp();
  }
```

```
  async initializeApp() {
    await this.checklistService.load();

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#2dd36f" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

Notice that we `await` the `load()` method before running the code to hide our `SplashScreen`. We don't *have* to use `await` here, we could just trigger a load and then hide the splash screen without waiting for the load to finish, but this way, all of the data will be loaded in and ready to go before the splash screen is removed. This won't add any significant delay, and might make the load look a bit smoother (rather than having data pop in shortly after the splash screen is removed).

That's all there is to it, the data will now be saved to the SQLite database whenever changes are made, and when the application is reopened all of the data will be loaded back in. Try adding some checklists or modifying the state of your checklists and reloading the application to see if the changes stick around.

Lesson 7: Creating an Introduction Slider and Theme

In the last lesson for the Quick Lists application, we are going to be adding a few final touches to improve the user experience. We will add a slideshow tutorial to show the user how to use the app (which will only display on their first time using the app) and we will also add some styles to make the application look a bit prettier.

Let's start off with the slider.

Slider Component

It's pretty common for mobile applications to display some instructions to the user through the use of a sliding card style tutorial. Ionic has a slide component built-in so we are going to make use of that, and to make sure the user doesn't have to go through the tutorial every time, we will be keeping track of if they have already seen it or not.

The slider itself is going to be pretty simple, it will allow us to display a series of images and there will be a button to start using the application.

First, we're going to build the slider component and then we are going to look at how to integrate it into our application. We'll start by creating the template.

> Modify `src/app/intro/intro.page.html` to reflect the following

```
<ion-content>
```

```
  <ion-slides pager="true">
    <ion-slide>
      <img src="assets/images/slide1.png" />
    </ion-slide>

    <ion-slide>
      <img src="assets/images/slide2.png" />
    </ion-slide>

    <ion-slide>
      <img src="assets/images/slide3.png" />
    </ion-slide>

    <ion-slide>
      <img src="assets/images/slide4.png" />
    </ion-slide>
  </ion-slides>
</ion-content>

<ion-footer>
  <ion-button expand="full" color="success" routerLink="/"
routerDirection="forward">
    Start Using Quicklists <ion-icon name="chevron-forward-
outline"></ion-icon>
  </ion-button>
</ion-footer>
```

The first thing you might notice about this is that it doesn't contain a header or toolbar,

only the content area (and a footer in this case, to hold the button). It's not necessary to include the header on every page, and we do not want to display it here. The rest of the code is pretty simple, we use `<ion-slides>` with a `pager` attribute so that the page indicator will be displayed on the slides, and we use `<ion-slide>` to define each one of our slides.

In the code above, the user will first see a slide containing the **slide1** image, then when they swipe they will see **slide2** and so on until they reach the last slide.

We don't even need to define any logic in **intro.page.ts** for this, but we do still have a bit of a problem regarding logic. We want to display this tutorial to the user if it is the first time they are using the application, but we don't want to display it every time they open the application. To deal with this, we are going to add a flag in storage that will record whether or not they have already been shown the tutorial. Then, we will just check this value in our home page and redirect them to the tutorial if necessary.

We want to perform the check for this flag as soon as we hit the home page, so we are again going to utilise the `ngOnInit` lifecycle hook on the home page, and we are also going to use the `NavController` so we can force the user back to the intro page if necessary.

> Add an `@ionic/storage` import in `src/app/home/home.page.ts`:

```
import { Storage } from "@ionic/storage";
```

> Modify the `constructor` in `src/app/home/home.page.ts` to reflect the following:

```
  constructor(
    private checklistService: ChecklistService,
    private alertCtrl: AlertController,
    private storage: Storage,
    private navCtrl: NavController
  ) {}
```

> Modify the `ngOnInit` method in `src/app/home/home.page.ts` to reflect the following:

```
  async ngOnInit() {
    const introPreviouslyShown = await
this.storage.get("introShown");

    if (introPreviouslyShown === null) {
      this.storage.set("introShown", true);
      this.navCtrl.navigateRoot("/intro");
    }

    this.checklistService.getChecklists().subscribe((checklists)
=> {
      this.checklists = checklists;
    });
```

```
  }
```

We're importing and making use of the `Storage` API now. Previously, we have just been navigating between pages using `routerLink`, but in this case, we want to change the route programmatically. This allows us to check the `introShown` value in storage, and then redirect the user to the intro page if necessary by calling:

```
this.navCtrl.navigateRoot("/intro");
```

It is also important that we set the `introShown` flag in the code above as well, otherwise, the intro page will keep showing every time the user opens the application. Also notice that we have made the `ngOnInit` method `async` now, as we need to `await` the value returned from storage.

NOTE: If you would like to clear the storage from within your application for testing purposes, you can do so using `this.storage.clear()` wherever `storage` is available. You can also do this through the Chrome DevTools by going to **Application > Clear storage**.

Theme

As far as functionality in the application goes, we're 100% done. Now we're just going to add a bit of styling to the application to make it look quite a bit better than it currently does.

If you remember from the basics section, there are quite a few different ways we can add styles to the application. We will be adding specific styles to each of our components, we will be adding some generic styles in our global file, and we will be overriding some CSS variables.

> Modify `src/intro/intro.page.scss` to reflect the following:

```
ion-slides {
  height: 100%;
  --bullet-background: var(--ion-color-light);
  --bullet-background-active: #000;
}

ion-slide {
  background-color: var(--ion-color-primary);
  display: flex;
  align-items: center;
}

ion-footer ion-button {
  margin: 1px;
}
```

These styles will make the background colour of the slides green, and by using a `flex` layout we are easily able to align the images that will be contained inside of the slides to the center. Since we have added these styles specifically to the `.scss` file for the

IntroPage, they will only apply to that one page.

Notice that we are making use of our theme variables here with `var` e.g. `background-color: var(--ion-color-primary)`. This will use whatever is defined as Ionic's primary colour (blue by default). By using these theme variables we can keep the colours throughout our application more consistent and update them all in once place (which we will be doing in just a moment).

Let's move on to the home page now.

> Modify `src/home/home.page.scss` to reflect the following (delete the existing styles):

```
ion-item-sliding {
  background-color: var(--ion-color-light);
  border-radius: 5px;
  margin: 10px 0 10px 10px;
  box-shadow: -2px 4px 4px 0px rgba(235, 68, 90, 0.1);
  overflow: visible;
}

ion-item-options {
  padding-right: 5px;
}

ion-label {
  font-size: 1.2em;
```

```
  font-weight: bold;
  color: var(--ion-color-dark);
  padding-top: 10px;
  padding-bottom: 10px;
}

ion-item {
  border-radius: 5px;
  border-left: 4px solid var(--ion-color-tertiary);
  padding-right: 5px;
}

ion-item span {
  font-weight: 400;
  opacity: 0.3;
  margin-right: 10px;
}
```

We're not doing anything too crazy here, just adding a few tweaks to the margins, padding, colours, and shadows. Now let's do the same to the checklist page.

> Modify `src/checklist/checklist.page.scss` to reflect the following:

```
ion-header {
  background-color: var(--ion-color-primary);
}
```

```
ion-label {
  font-weight: bold;
  margin: 20px;
}
```

Once again, just a few minor tweaks here. Now we are going to add the styles that will apply across the entire application:

> Add the following style to the bottom of `src/global.scss`:

```
ion-title img {
  max-height: 30px;
  width: auto;
}

ion-content {
  --ion-background-color: var(--ion-color-primary);

  ion-item-sliding {
    --ion-background-color: var(--ion-color-light);
  }
}
```

Here we're just setting a maximum height for the logo, and a default background colour for

our content area and items. Notice that we aren't just modifying a CSS property for `ion-content` and `ion-item-sliding` we are changing the value of a CSS variable. This is an important distinction, we are **not** changing the `background-color` property for these elements, we are setting the value of a CSS variable that can be used if desired (and many Ionic components will make use of this `--ion-background-color` variable to set their background colour).

This means anywhere in the application that tries to use the `--ion-background-color` CSS variable will use our modified value above instead. Notice that we are actually modifying the same CSS variable twice. We want the `ion-content` component and anything inside of it to use a value of `--ion-color-primary` but for any `ion-item-sliding` components inside of `ion-content` (and anything else inside of those sliding items) we want to use a value of `--ion-color-light` for the background.

Finally, we are going to modify the CSS variables for our theme. I have already created a theme for us to use, but a great way to generate your own themes is to use the Ionic [Color Generator]() tool.

> Modify the `:root` selector in `src/theme/variables.scss` to reflect the following:

```
:root {
  --ion-color-primary: #32db64;
  --ion-color-primary-rgb: 50, 219, 100;
  --ion-color-primary-contrast: #000000;
  --ion-color-primary-contrast-rgb: 0, 0, 0;
  --ion-color-primary-shade: #2cc158;
```

```
  --ion-color-primary-tint: #47df74;

  --ion-color-secondary: #2dd36f;
  --ion-color-secondary-rgb: 45, 211, 111;
  --ion-color-secondary-contrast: #000000;
  --ion-color-secondary-contrast-rgb: 0, 0, 0;
  --ion-color-secondary-shade: #28ba62;
  --ion-color-secondary-tint: #42d77d;

  --ion-color-tertiary: #199e41;
  --ion-color-tertiary-rgb: 25, 158, 65;
  --ion-color-tertiary-contrast: #ffffff;
  --ion-color-tertiary-contrast-rgb: 255, 255, 255;
  --ion-color-tertiary-shade: #168b39;
  --ion-color-tertiary-tint: #30a854;

  --ion-color-success: #2dd36f;
  --ion-color-success-rgb: 45, 211, 111;
  --ion-color-success-contrast: #ffffff;
  --ion-color-success-contrast-rgb: 255, 255, 255;
  --ion-color-success-shade: #28ba62;
  --ion-color-success-tint: #42d77d;

  --ion-color-warning: #ffc409;
  --ion-color-warning-rgb: 255, 196, 9;
  --ion-color-warning-contrast: #000000;
  --ion-color-warning-contrast-rgb: 0, 0, 0;
  --ion-color-warning-shade: #e0ac08;
  --ion-color-warning-tint: #ffca22;
```

```
  --ion-color-danger: #eb445a;
  --ion-color-danger-rgb: 235, 68, 90;
  --ion-color-danger-contrast: #ffffff;
  --ion-color-danger-contrast-rgb: 255, 255, 255;
  --ion-color-danger-shade: #cf3c4f;
  --ion-color-danger-tint: #ed576b;

  --ion-color-dark: #1c1d3b;
  --ion-color-dark-rgb: 28, 29, 59;
  --ion-color-dark-contrast: #ffffff;
  --ion-color-dark-contrast-rgb: 255, 255, 255;
  --ion-color-dark-shade: #191a34;
  --ion-color-dark-tint: #33344f;

  --ion-color-medium: #f4f5f8;
  --ion-color-medium-rgb: 244, 245, 248;
  --ion-color-medium-contrast: #000000;
  --ion-color-medium-contrast-rgb: 0, 0, 0;
  --ion-color-medium-shade: #d7d8da;
  --ion-color-medium-tint: #f5f6f9;

  --ion-color-light: #ffffff;
  --ion-color-light-rgb: 255, 255, 255;
  --ion-color-light-contrast: #000000;
  --ion-color-light-contrast-rgb: 0, 0, 0;
  --ion-color-light-shade: #e0e0e0;
  --ion-color-light-tint: #ffffff;
}
```

Ionic also comes with styling for a **dark mode** by default, which will be enabled if the users preferences are set to use dark mode where available. We will be removing that, as it will mess with the styles that we have set up above.

> Remove everything inside of `@media (prefers-color-scheme: dark) { }` in `src/theme/variables.scss`:

Alternatively, you could supply your own different set of styles that you want to be activated when dark mode is switched on. I have this as a homework exercise in the next lesson, so if you would like to have a go at implementing your own dark mode theme you should keep the current contents of `@media (prefers-color-scheme: dark) { }` so that you have a base to work from.

One of the coolest things about Ionic is how well it handles UI differences between iOS and Android, for the most part, it just works flawlessly out of the box. If you take a look at the application on iOS and on Android, you will see the differences:

NOTE: A handy way to see iOS and Android side by side is to use Ionic Lab, which can be activated by using the `ionic serve -l` command (you will prompted to install `@ionic/lab`).

As you can see, Ionic automatically conforms to the norms of whatever platform the application is running on.

Summary

That's it! The application should now be completely finished and it finally actually looks pretty nice too.

Conclusion

Congratulations on making it through the Quick Lists lessons. This application is a great example for beginners to start getting their feet wet, and the main takeaways from it are:

- How to create, read, update and delete data
- How to permanently store data and retrieve it
- How to create and use your own service effectively
- How to navigate and pass data between pages
- How to style/theme an application

There's always room to take things further though, especially when you're trying to learn something. Following tutorials is great, but it's even better when you figure something out for yourself. Hopefully, you have enough background knowledge now to start trying to extend the functionality of the application by yourself, here are a few ideas to try out (but feel free to improvise):

- Implement your own dark mode theme using `@media (prefers-color-scheme: dark) { }` in `src/theme/variables.scss` **[EASY]**
- Add a `dateCreated` field to the checklists that records when a checklist was created, and display it in the template **[MEDIUM]**
- Figure out how many items have been marked as completed in a single checklist, and display a progress indicator (e.g. 5/7 completed) **[MEDIUM]**
- Add the ability to attach additional notes to any specific checklist item. Decide how/when to best display those additional notes. **[HARD]**

Remember, the Ionic documentation is your best friend when trying to figure things out.

What next?

You have a completed application now, but that's not the end of the story. You also need to get it running on a real device and submitted to app stores, which is no easy task. The final sections in this book will walk through how to take what you have done here, and get it running on a real device, as well as publishing it. If this is something that you are interested in learning now, make sure to check those sections out.

Giflist

Lesson 1: Giflist Introduction

Giflist was the first application I created when the Ionic 2 alpha was announced (for those of you who haven't been using Ionic for that long, Ionic 2 was when Ionic switched from the original AngularJS framework to the modern Angular that we use today - this was a pretty big deal at the time). It has been updated a lot since then, including for the latest version of Ionic and Capacitor for the current version of this book. Even though it was the first app I built with the modern version of Ionic, it's still one of my favourites, so I'm excited to walk you through building it.

I think it's one of my favourites because it's just a fun/entertainment oriented app. It's a fun app to build because a lot of interesting topics are covered like using the Reddit API and using HTML5 video, and it's a fun app to use because... who doesn't like looking at cute, funny, or awesome GIFs?

About Giflist

Giflist has a pretty simple concept, the general idea is that a user can enter in any **subreddit** from reddit, and the app will fetch and display GIFs from that subreddit.

I'm going to assume if you're reading this book then you know what Reddit is - but - I don't really like to make any assumptions so if you're somehow *not* familiar with Reddit, basically it's a site where people can submit links to anything and the user base votes up

stuff that they think is cool or relevant. A "subreddit" is basically a category on Reddit, a few popular ones being:

- gifs
- askreddit
- worldnews

As well as just fetching GIFs from Reddit, users will also be able to supply some settings to configure the application to their preferences. Although the concept is pretty simple, there are a few interesting things that you will learn through building it, this will include:

- Fetching data from a 3rd party API
- Creating advanced interfaces
- Manipulating data
- Data storage
- Using Observables
- Themes
- Lists
- Modals
- HTML5 Video

Here's a quick run down on the exact features of the application:

- The user will be able to input any subreddit
- An endless list of GIFs will be displayed as a list (assuming GIFs are available)
- The user will be able to play a GIF by tapping it
- The user will be able to view the original submission on reddit through an `InAppBrowser` instance

- The user will be able to set the following options: *default subreddit*, *sort order*, *GIFs per page*
- The user's settings will be remembered upon returning to the application

and a couple of screenshots to put it into context:

Settings
animalsbeingderps.
To play a GIF, just tap it. To view the original submission for any GIF, tap the title section and it will open in reddit. You can configure some settings for the application below. Enjoy!
Default Subreddit
gifs
Sort
Hot
New
Page Size
10
20
30
Apply Settings

animalsbeingderps
Not quite ready for bedtime
86
Useless hooman, belly ain't gona rub itself.

Lesson Structure

1. Getting Ready
2. The List Page
3. The Reddit API and HTML5 Video
4. Integrating the Settings Service
5. The Settings Page
6. Styling

Ready?

Now that you know what you're in for, let's get to building it!

Lesson 2: Giflist Getting Ready

In this lesson, we are going to prepare our application for the journey ahead. We are going to generate the application, and we are also going to set up all of the components, plugins, and routes that we need. The idea is to get all of the general scaffolding required for most projects out of the way so we don't have to mess around with creating files and configuring things throughout the rest of the lessons.

At the end of this first lesson, we should have a nice skeleton application set up with everything we need to start diving into coding.

A good rule of thumb before starting any new application is to make sure you have the latest version of the Ionic CLI installed, so if you haven't done it recently then make sure to run:

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

If your environment is not set up correctly, you may run into permission errors when installing this package globally. If you are having permission issues, you can use the Ionic teams guide to solving these issues: Resolving permission errors.

If you do not already have a recent version of Node/npm set up on your machine, make sure to complete the Environment Setup guide first.

Generate a new application

We will be using the blank starter template for this application which, as the name implies, is basically an empty Ionic project. It comes with one page built-in called **HomePage** which we will reuse as our main GIFs page, but we will also be adding an additional page.

> Run the following command to generate a new application

```
ionic start giflist blank --type angular
```

When you are prompted with the following:

```
? Integrate your new app with Capacitor to target native iOS and
Android? (y/N)
```

Say **Yes**, as we will be utilising Capacitor for native builds. If you do not set up Capacitor when first creating your application, you can always do it manually later as well by running the following command:

```
ionic integrations enable capacitor
```

NOTE: You should still install Capacitor in this project even if you do not intend to build for iOS or Android, as we will still be using Capacitor for the web platform as well.

> Make the new project your current working directory by running the following command:

```
cd giflist
```

Your project should now be generated and you can open up the project folder in your favourite editor (I recommend using Visual Studio Code). You can take a look at the application as it is now by running the following command:

```
ionic serve
```

which for now should look something like this:

Blank

Ready to create an app?

Start with Ionic UI Components

Create the Required Components

The structure of this application is pretty simple, so all we're going to have is two pages: the list page and the settings page. We've already got a **HomePage** component which we will use as our list page, so we only need to create the **Settings** page.

NOTE: You will need to stop serving your application in order to run these commands. If your application is currently being served through the command line, make sure to hit `Ctrl + C` first to stop the process.

> Run the following command to generate the Settings page:

```
ionic g page Settings
```

Create the Required Services

As well as our two pages, we are also going to create a settings service to handle storing and retrieving the user's settings, and a service to help us retrieve (and parse) data from Reddit.

> Run the following command to generate a Settings service:

```
ionic g service services/Settings
```

NOTE: Notice that we use `services/Settings` instead of just `Settings` this time. This is because we want to generate all of our services inside of a folder called `services`. You

could also use this same technique for pages if you wanted to store all of your pages inside of a specific folder like `pages`.

> Run the following command to generate a Reddit service:

```
ionic g service services/Reddit
```

Create the Interfaces

We will eventually be defining a custom interface to represent our `Gif`, `RedditPost`, and `Settings` data types. There is no generate command for this, so you will need to create it manually.

> Create a new folder inside of the `app` folder called `interfaces`

> Create a new file inside of the interfaces folder called `gif.ts`

> Create a new file inside of the interfaces folder called `reddit-post.ts`

> Create a new file inside of the interfaces folder called `settings.ts`

Configure the Routes

As we discussed in the basics section, we need to define routes so that our application can match up the current URL to a particular component that you want to display. Generating the application and using the generate commands do most of the work for us, as routes are automatically injected into the **app-routing.module.ts** file. However, we will

need to make some slight modifications to these.

> Modify `src/app/app-routing.module.ts` to reflect the following:

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

const routes: Routes = [
  {
    path: "home",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
  },
  {
    path: "",
    redirectTo: "home",
    pathMatch: "full",
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Notice that we do not have a route for the **Settings** page? This is because we will not be using the `<ion-router-outlet>` to display our settings page - we will be launching it as a modal inside of our home page.

IMPORTANT: You will need to delete the **src/app/settings/settings.module.ts** file, otherwise, you will get errors when building your application. Since we are using the component directly as a modal, it does not need a module file.

Configure Capacitor

At this point, you should already have Capacitor added to your project. However, if you did not respond **yes** to the prompt about installing Capacitor when initially creating the project, you should manually set it up now.

> Run the following command (if required) to enable Capacitor:

```
ionic integrations enable capacitor
```

Keep in mind that by default your application will be given a **Bundle ID** of `io.ionic.starter`. This Bundle ID is used to identify your native iOS/Android builds, and you will eventually want to change this to something unique to you or your company. You can change it later if you wish, but it is easier to do before we add the iOS and Android platforms. You may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`. If you don't

intend on eventually modifying and publishing this project, then this step doesn't matter.

If you look at the **capacitor.config.json** file, you will notice that the `webDir` is listed as `www`. This is the folder that Capacitor will look to when it is copying over the code for your application into the native projects. It doesn't care about the rest of your source code, it only cares about the built output (i.e. the code that is actually run through the browser). By default, the `www` folder will not exist in your project until you perform a build of your application manually. We are going to do that now.

> Run the following command to generate a build of your application:

```
ionic build
```

It is good to use the standard `ionic build` command whenever you want to test your application on a device during development, but keep in mind that this creates a development build. When you are building the final version of your application, or if you want to test a production version of your application, you should run:

```
ionic build --prod
```

which will create an optimised production build.

Add Native Platforms

By default, the iOS and Android platforms will not be enabled in your project. If you want to add these platforms you can do so now (or you can do it later if you prefer). Remember, building for iOS will require Xcode and building for Android will require Android Studio - if you want to add these platforms now, you should make sure you have followed the instructions in the Generating an Ionic Application lesson to install Xcode, Android Studio, and their dependencies.

```
ionic cap add ios
```

Keep in mind that the first time the `Updating iOS native dependencies` line is run it may take quite a while.

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your Ionic application code and dependencies at any time by running:

```
ionic cap sync
```

IMPORTANT: If you want to use a production build with Capacitor, you will need to add the `--prod` flag to the sync command, e.g: `ionic cap sync --prod`.

When you make changes to your application build, you will need to run this command to copy the changes over to the native iOS/Android projects created by Capacitor. If you do not run this command, any changes you make to your Ionic application will not be present when running the application on iOS/Android.

Set up the Root Component

The default starter templates for Ionic utilise **Ionic Native** to handle hiding the **Splash Screen** and styling the **Status Bar**. Whilst you can still use **Ionic Native** and **Cordova** plugins inside of a Capacitor project, Capacitor provides this functionality through default APIs, so we are going to use those instead. It is also important to remember that the Cordova Splash Screen plugin is not compatible with Capacitor, so make sure that you do not install it in your project.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
```

```
  constructor() {
    this.initializeApp();
  }

  async initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#e74c3c" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

You can also use the `StatusBar` API to hide the status bar entirely, but in this case we are styling it with a colour that will match our application. The status bar is the bar at the top of native applications that display the time, battery level, etc.

Set up Ionic Storage

We will be making use of the Ionic Storage API in this application. Ionic provides a simple key/value storage API that we can use in our applications to store data - it provides a consistent API and will automatically use the best storage mechanism available. That means that if we install the SQLite plugin in our project, it will store our data in native storage (rather than browser local storage which is more likely to get wiped by the

operating system randomly). If native storage is not available, it will default back to browser storage.

To enable Ionic storage, you need to install it.

> Install Ionic Storage with the following command:

```
npm install @ionic/storage
```

and you will also need to add it to your root module file.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
```

```
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported `IonicStorageModule` and we have also added it to the `imports` array. We have also removed the `SplashScreen` and `StatusBar` providers from Ionic Native since we are not using them anymore.

Set up Additional Modules

As well as the Ionic Storage module, we will also need to set up a couple more modules in this application. First, since we are making use of HTTP requests, we will need to add the `HttpClientModule` to the root module in our application.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { HttpClientModule } from "@angular/common/http";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";
```

```
import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [
    BrowserModule,
    HttpClientModule,
    IonicModule.forRoot(),
    IonicStorageModule.forRoot(),
    AppRoutingModule,
  ],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

We will also be making use of advanced Angular form features, so we will need to include the `ReactiveFormsModule` as well as the regular `FormsModule` in the module for our Home page. Since we will be launching the Settings page as a modal inside of our home page, we will also need to import the `SettingsPage` here and add it to the `declarations` array.

> Modify `src/app/home/home.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { IonicModule } from "@ionic/angular";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { HomePage } from "./home.page";
import { SettingsPage } from "../settings/settings.page";

import { HomePageRoutingModule } from "./home-routing.module";

@NgModule({
  imports: [CommonModule, FormsModule, ReactiveFormsModule,
IonicModule, HomePageRoutingModule],
  declarations: [HomePage, SettingsPage],
})
export class HomePageModule {}
```

Set up Plugins

We are going to set up any native plugins that we need now. Remember that when installing Cordova plugins in a Capacitor project you should install them using the `npm install` command, **not** with `ionic cordova plugin add`.

> Run the following command to add the SQLite plugin:

```
npm install cordova-sqlite-storage
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

Remember, after making changes to the native plugins in your project you will need to run:

```
ionic cap sync
```

This will copy the new plugins you have installed over to the native projects.

Summary

That's it! We're all set up and ready to go, now we can start working on the interesting stuff. Make sure to test your application by running `ionic serve` every now and then, it's easier to catch and fix errors along the way than waiting until later. If you ever get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again.

Lesson 3: The List Page

In the last lesson we worked on getting everything set up correctly, and now we're going to start actually building stuff. In this lesson, we'll mostly be focusing on modifying the **Home** page to act as the list that will contain our feed of GIFs.

The Reddit Service

The layout that we are about to create relies heavily on pulling in data from Reddit, and we will create a service to handle these operations for us. We're not going to jump right into implementing the nitty-gritty details of the Reddit service just yet, but we do want to set up the functions we will eventually create so that we will be able to reference them in the rest of this lesson. We will just create a basic skeleton implementation of the service now, and finish implementing it later.

> Modify `src/app/services/reddit.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { BehaviorSubject, Observable } from "rxjs";
import { map, takeWhile } from "rxjs/operators";

import { SettingsService } from "../services/settings.service";
import { RedditPost } from "../interfaces/reddit-post";
import { Gif } from "../interfaces/gif";
```

```
import { Settings } from "../interfaces/settings";

@Injectable({
  providedIn: "root",
})
export class RedditService {
  constructor(private http: HttpClient, private settingsService:
SettingsService) {}

  public loading: boolean = false;
  private loading$: BehaviorSubject<boolean> = new
BehaviorSubject<boolean>(false);

  private gifs: Gif[] = [];
  private gifs$: BehaviorSubject<Gif[]> = new
BehaviorSubject<Gif[]>([]);

  public noPostsRemaining: boolean = false;
  private cancelSearch: boolean = false;
  private settings: Settings;
  private after: string;
  private currentPage: number = 1;
  private timesRetried: number = 0;

  getGifs(): Observable<Gif[]> {
    return this.gifs$;
  }

  getLoadingState(): Observable<boolean> {
```

```
    return this.loading$;
  }

  loadGifsFromActiveSubreddit(): void {}

  getBestSrcForGif(post: RedditPost): string | boolean {
    return false;
  }

  nextPage(): void {}

  async reset(subreddit?: string): Promise<void> {
    return Promise.resolve();
  }
}
```

This is the basic skeleton for the service. It defines a bunch of member variables and methods that will handle grabbing the data from Reddit. The `loadGifsFromActiveSubreddit` method will be the most significant method in this service, followed by the `getBestSrcForGif` method which will handle sorting through the data returned from Reddit and finding the best available media. What the member variables that we have defined do may not be immediately obvious, so let's go through them (in order):

- The `loading` boolean will indicate whether data is currently being loaded from Reddit, and the `loading$` observable/behavior subject will allow us to subscribe to status updates to this `loading` boolean. This is especially useful when we need to

wait for the loading status to change before performing some operation. If you are unfamiliar with `BehaviorSubject` I would recommend revisiting the **Observables/BehaviorSubject** section from **Lesson 4** of the **QuickLists** application.

- The `gifs` array is used to store all of the gifs that we are going to display to the user. This array is used internally in this service to manage the data, and then we also have a `gifs$` observable/behavior subject that will allow other parts of the application to subscribe for changes to these gifs
- The `noPostsRemaining` boolean is used as a flag to indicate when the Reddit API is no longer able to return any more posts
- The `cancelSearch` boolean will break out of the recursive calls to `loadGifsFromActiveSubreddit()` (which will run repeatedly until enough gifs have been found to fill a page) if it is set to `true`. This will allow a search in progress to be cancelled if the user wants to change subreddits midway through a search
- We have a `settings` object that will allow the user to define specifically how the posts should be retrieved from Reddit. This data will be pulled in from the `Settings` service
- The `after` string keeps track of the `name` of the last GIF that we pulled in from the Reddit API. This is important as we can use this to say to the Reddit API "give us the posts that come *after* this particular post". This allows us to paginate our results.
- The `currentPage` number will keep track of the page that we are on currently because we will be allowing users to load in more GIFs as they scroll down the page. In this sense, each time the user loads in a new set of data this will be a "page" (even though it will actually just be an infinite list).
- The `timesRetried` is a bit of an interesting one. You will learn more about this later, but basically, our service will keep trying to pull in data from Reddit until it has enough GIFs. This variable is a way for us to specify how many times we want to the

service to keep trying to fetch data if it doesn't get enough before giving up. If a subreddit does not contain many gifs, then we may never find enough to fill a page.

As well as setting up our member variables, we are also importing the **HttpClient** so that we can launch HTTP requests, and we are importing our `Settings` service (which we are yet to implement). We are also importing the `map` and `takeWhile` operators from RxJS, but we are going to talk about that a little later.

We will finish up this service later, but for now, we are going to move onto creating the home page layout. Before we do that, we also need to define the interfaces that we are referencing in this service.

Defining the Interfaces

You will notice in the service above that we are using custom `Gif`, `Settings`, and `RedditPost` interfaces. Our `Settings` interface is not really any different to what we have been doing with custom types/interfaces so far, so let's set that one up first.

> Modify `src/app/interfaces/settings.ts` to reflect the following:

```
export interface Settings {
  perPage: number;
  subreddit: string;
  sort: "hot" | "new";
}
```

This will enforce the properties that will be allowed to define the user's settings. We also need to define our `RedditPost` interface that will define the "shape" of the response we will be getting back from Reddit, and our `Gif` interface that will define what a gif object will look like in our application once we have parsed the data returned from reddit and retrieved the *specific* data we want from it (the Reddit API returns *a lot* of data, most of which we don't need).

The interface we are about to create for `RedditPost` will look absurdly complicated but it is worthwhile to do (and isn't that difficult to create once you understand what is going on). To highlight why this is so useful, take a look at one of the properties we will be accessing from the posts returned back from Reddit:

```
post.data.secure_media.reddit_video.fallback_url;
```

We don't need a a type/interface to access this data, we could just use the `any` type for example and access whatever properties we want on the `post` object. However, there are two main benefits to taking the time to create a custom interface to add type safety to the `post` object:

- If we make a typo anywhere, TypeScript will warn us about it. If I accidentally used `redditVideo` instead of `reddit_video` I would no about it immediately, rather than it introducing a bug into the application
- If we are using something like Visual Studio Code, our text editor will provide auto-completion hints that will allow us to quickly browser through all of the available properties of `post`. If we didn't set this up, then we would constantly have to keep

referring back to the raw response data from Reddit to figure out what various properties are called.

With that justification out of the way, let's implement it.

> Modify `src/app/interfaces/reddit-post.ts` to reflect the following:

```
export interface RedditPost {
  data: RedditPostData;
}

interface RedditPostData {
  author: string;
  name: string;
  permalink: string;
  preview: RedditPreview;
  secure_media: RedditMedia;
  title: string;
  media: RedditMedia;
  url: string;
  thumbnail: string;
  num_comments: number;
}

interface RedditPreview {
  reddit_video_preview: RedditVideoPreview;
}
```

```
interface RedditVideoPreview {
  is_gif: boolean;
  fallback_url: string;
}

interface RedditMedia {
  reddit_video: RedditVideoPreview;
}
```

The response we will get back from Reddit will contain a `data` object. That `data` object will then contain various properties. Some of those properties will also be additional objects. This is why this interface looks so complicated. Each time a property is an object, we need to create another interface to define the shape of that object.

You can create this interface quite simply/methodically. For example, since the response returned from Reddit just contains a `data` object I would start defining this interface like this:

```
export interface RedditPost {
  data: any;
}
```

Using an `any` type here is mostly useless, as it won't give us any detail about what is contained within that `data` object. I would then look at the `data` object that Reddit

returned, and define a new interface to capture that:

```
export interface RedditPost {
  data: RedditPostData;
}

interface RedditPostData {
  author: string;
  name: string;
  permalink: string;
  preview: any;
  secure_media: any;
  title: string;
  media: any;
  url: string;
  thumbnail: string;
  num_comments: number;
}
```

However, this interface also contains three more objects: `preview`, `secure_media`, and `media`. Again, I could just use `any` for those but it is not as useful as creating a proper interface for them. So then I would go ahead and look at the Reddit data again to create interfaces for each of those. You can continue in this fashion until you are satisfied with the level of type detail that you have.

It is also worth noting that we have only bothered to include types for the data that we are

actually interested in. The actual response from Reddit contains maybe 10x more data than what we have captured above. However, this works to our advantage, because now we can just decide what we are interested in once rather than having to look through every single property returned from Reddit every time.

Let's define our `Gif` interface now.

> Modify `src/app/interfaces/gif.ts` to reflect the following:

```
export interface Gif {
  src: string | boolean;
  author: string;
  name: string;
  permalink: string;
  title: string;
  thumbnail: string;
  comments: number;
  loading?: boolean;
  dataLoaded?: boolean;
}
```

We have a separate type for `Gif` instead of just using `RedditPost` because there are inconsistencies in the data returned from Reddit which we will discuss later, and we also require some additional fields to display gifs properly in our template.

The Layout

Before we start building the layout for our home page, let's take a look at what we're building:

animalsbeingderps
Not quite ready for bedtime
86
Useless hooman, belly ain't gona rub itself.

It's a reasonably simple layout, we have a toolbar at the top that contains a search bar and a settings button (which will launch our Settings page later). Beneath that, we have a list that holds all of the gifs returned from Reddit. Not pictured is the `<ion-infinite-scroll>` which sits at the bottom of the list which will load in the next page of gifs when it is reached.

First, we're going to look at the entire template to see everything in context, then we're going to break it down into smaller chunks that we will discuss in detail.

> Modify `src/app/home/home.page.html` to reflect the following

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-searchbar animated placeholder="subreddit..."></ion-
searchbar>
    <ion-buttons slot="end">
      <ion-button (click)="openSettings()">
        <ion-icon slot="icon-only" name="settings"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-list lines="none">
    <div class="gif">
```

```
      <ion-item button detail="false"> GIF GOES HERE </ion-item>
      <ion-list-header style="text-align: left">
        <ion-label>TITLE GOES HERE</ion-label>
      </ion-list-header>
    </div>
  </ion-list>

  <ion-infinite-scroll *ngIf="gifs.length > 0" threshold="100px"
(ionInfinite)="loadMore($event)">
    <ion-infinite-scroll-content loadingSpinner="bubbles"
loadingText="Fetching gifs...">
    </ion-infinite-scroll-content>
  </ion-infinite-scroll>

  <ion-item lines="none" *ngIf="redditService.noPostsRemaining">
    <ion-label style="text-align: center; font-weight: bold">Can't
find any more gifs!</ion-label>
  </ion-item>
</ion-content>
```

In the first part of this code we are setting up the header:

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-searchbar animated placeholder="subreddit..."></ion-
searchbar>
```

```
    <ion-buttons slot="end">
      <ion-button (click)="openSettings()">
        <ion-icon slot="icon-only" name="settings"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

We add the `primary` color attribute to the `<ion-toolbar>` which will alter its style to use the `--ion-color-primary` colour, which we will be changing later.

Rather than having a separate toolbar to contain our searchbar, we are using our main application toolbar to hold the `<ion-searchbar>`. We also use `<ion-buttons>` to place our settings button, which will launch our settings page. Using the `end` slot will place the button on the right, and if we wanted to place a button on the left we could use the `start` slot instead. We've also added a `(click)` listener to this button so that the `openSettings` method will be called when the button is clicked. We haven't created this function yet so nothing will happen, but we will define it later in **home.page.ts**.

Next, we define a list in the main content area:

```
<ion-list lines="none">
  <div class="gif">
    <ion-item button detail="false"> GIF GOES HERE </ion-item>
    <ion-list-header style="text-align: left">
```

```
      <ion-label>TITLE GOES HERE</ion-label>
    </ion-list-header>
  </div>
</ion-list>
```

Lists are one of the most frequently used components in mobile applications. In Ionic, you can create an `<ion-list>` and supply it any number of `<ion-item>` tags to create a list. For now, we just have a single item, but later we will modify this to automatically loop over every gif that we want to display. Notice that we are also using `<ion-list-header>` to create a header area where we will be able to display the title of the gif, and we also use `lines="none"` so that there are no borders around list items. The reason that we have surrounded the `<ion-item>` and `<ion-list-header>` in a `<div>` is because eventually, that is what we are going to use our `*ngFor` loop with. We don't want to just repeat the `<ion-item>` for each one of our gifs, we want to repeat both the `<ion-item>` and the `<ion-list-header>` for each item. By adding a `<div>` to contain both of those, we can then easily just loop that `<div>` with `*ngFor`.

The last bit of code we have in our template is to handle the infinite loading

```
<ion-infinite-scroll *ngIf="gifs.length > 0" threshold="100px"
(ionInfinite)="loadMore($event)">
  <ion-infinite-scroll-content loadingSpinner="bubbles"
loadingText="Fetching gifs...">
  </ion-infinite-scroll-content>
</ion-infinite-scroll>
```

```
<ion-item lines="none" *ngIf="redditService.noPostsRemaining">
  <ion-label style="text-align: center; font-weight: bold">Can't
find any more gifs!</ion-label>
</ion-item>
```

Nothing too much is happening here, we are just using the standard `<ion-infinite-scroll>` behaviour. We will implement the rest of this later, but the basic idea is that when the user reaches the bottom of the list the `<ion-infinite-scroll>` will automatically call the `loadMore` method so that we can fetch more gifs to load in at the bottom of the list. Also notice that we only display the infinite scroll if `gifs.length` is greater than `0`. This is to stop it from displaying when the gifs are first being loaded in.

We also put an extra `<ion-item>` at the bottom of the page which will display a notice if there are no posts left to return from the Reddit API.

The Home Page Class

With the template defined we have our "view" sorted, now we need to create the class definition to handle all the logic our list page will use. This is where we will define all the functions that our template references, as well as any other code we want to run on this page.

Again, we are going to set up a skeleton for the entire class first and then we will discuss it bit by bit.

> **Modify `src/app/home/home.page.ts` to reflect the following:**

```
import { Component, OnInit, ViewChild } from "@angular/core";
import { FormControl, FormGroup } from "@angular/forms";
import { ModalController, IonContent } from "@ionic/angular";
import { Plugins } from "@capacitor/core";

import { SettingsPage } from "../settings/settings.page";
import { RedditService } from "../services/reddit.service";

import { debounceTime, distinctUntilChanged, takeWhile, skip }
from "rxjs/operators";
import { Gif } from "../interfaces/gif";

const { Browser } = Plugins;

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage implements OnInit {
  @ViewChild(IonContent, { static: false }) contentArea:
IonContent;

  public gifs: Gif[];
  public subredditForm: FormGroup;
```

```
  constructor(public redditService: RedditService, private
modalCtrl: ModalController) {
    this.subredditForm = new FormGroup({
      subredditControl: new FormControl(""),
    });
  }

  ngOnInit() {}

  showComments(gif: Gif): void {}

  async openSettings(): Promise<void> {}

  playVideo(ev: Event, gif: Gif): void {}

  loadMore(ev: Event): void {}
}
```

There's obviously still quite a bit of code that needs to be added to this class, but even the basic setup of the class we've just added is looking pretty complicated. Let's start talking through it.

First up are our **import** statements:

```
import { Component, OnInit, ViewChild } from "@angular/core";
import { FormControl, FormGroup } from "@angular/forms";
import { ModalController, IonContent } from "@ionic/angular";
```

```
import { Plugins } from "@capacitor/core";

import { SettingsPage } from "../settings/settings.page";
import { RedditService } from "../services/reddit.service";

import { debounceTime, distinctUntilChanged, takeWhile, skip }
from "rxjs/operators";
import { Gif } from "../interfaces/gif";

const { Browser } = Plugins;
```

There are a bunch of imports here because we are setting up everything we will use later, but this is still probably quite a few more imports than you will likely use in most of your classes.

The first few imports are pretty standard - we will be launching our settings page as a modal so we import the **ModalController**, and we also import both of our own **RedditService**. We also import **ViewChild** and **IonContent** because we will be creating a reference to the `<ion-content>` component so that we can call its `scrollToTop` method.

We will be creating a more advanced type of form input on this page, which is why we are importing **FormGroup** and **FormControl**. The **FormGroup** allows us to create a group of form controls, and the **FormControl** will allow us to create a form control for the inputs which will give us access to an Observable to observe changes to the input. We also import some RxJS operators that we will be making use of in conjunction with our fancy form control. Since our form control gives us an observable to work with, we can use these

RxJS operators to manipulate that observable - we will talk more about these later.

We also set up a reference for the `Browser` Capacitor plugin that we will be using from the **Plugins** import.

In the next section of code, we have our `constructor` function and the `ngOnInit` lifecycle hook. The constructor is an important part of the class because it is the first bit of code that will be executed when the component is created. It allows us to inject and set up references to any components or services we are using and it's also a good spot to make any function calls that you need to execute right away. Keep in mind that it is generally considered best practice to avoid doing "work" in the `constructor`, so you may prefer to instead place code that you need to execute right away inside of the `ngOnInit()` lifecycle hook (this will also execute when the page is first loaded). Avoid adding code to your constructor wherever possible, but sometimes it is the most practical place to execute something.

```
  public subredditForm: FormGroup;

  constructor(public redditService: RedditService, private
modalCtrl: ModalController) {
    this.subredditForm = new FormGroup({
      subredditControl: new FormControl(""),
    });
  }

  ngOnInit() {}
```

Above the `constructor` we set up a member variable that we will bind to our search input. Since this `FormGroup` will be tied to our searchbar, we need to immediately instantiate it in our `constructor`. We create a new `FormGroup` with one `FormControl` called `subredditControl`. Later, we will attach this control to the searchbar in our template. Once we attach the form control to the search bar input, we can start watching it for changes. If all of this `FormGroup` and `FormControl` shenanigans is feeling a little weird to you, make sure to read or re-read the **User Input** lesson from the basics section of the book.

Let's take a look at the remaining methods in the class:

```
showComments(gif: Gif): void {}

async openSettings(): Promise<void> {}

playVideo(ev: Event, gif: Gif): void {}

loadMore(ev: Event): void {}
```

The rest of these methods will be called from somewhere in our template, but we will be implementing these later.

Using an Observable to Control Searching

We're going to get a bit fancy now and make use of **Observables**. We've discussed what exactly an Observable is in the basics section of this book so if you can't quite remember

I'd recommend going back and taking a look at the **Fetching Data, Observables and Promises** section.

Most of the time when using Observables you will simply be subscribing to some Observable that is returning values for you, like when using the **HttpClient** service to fetch some data (which we will be doing for this application in the next section). You rarely *need* to create or manipulate observables yourself, but they do offer a lot of powerful functionality should you familiarise yourself with how they work.

We are using the **FormControl** service we imported before to create a `FormControl` that will supply an observable to us. Form Controls work very similarly to two-way data binding with `[(ngModel)]` in that it ties a variable in the class definition, to an input field in the template. Since we have already created the `subredditForm` member variable, which contains our `FormControl`, all we need to do is add it to the searchbar in the template.

> Modify the searchbar in `src/app/home/home.page.html` to reflect the following:

```
<form [formGroup]="subredditForm">
  <ion-searchbar
    animated
    placeholder="subreddit..."
    formControlName="subredditControl"
    value=""
  ></ion-searchbar>
</form>
```

What we've done here is surround our searchbar in a form that has a `formGroup` property that links to the `FormGroup` we created. We've also added a `formControlName` which matches up to the form control we created inside of the form group in our `constructor`.

With all of this setup, we will be able to use this form control to watch for changes on the searchbar input. We are going to enable this behaviour by running some code inside of our `ngOnInit()` hook.

> Modify `ngOnInit` in `src/app/home/home.page.ts` to reflect the following:

```
  ngOnInit() {
    this.redditService.getGifs().subscribe((gifs) => {
      this.gifs = gifs;
    });

    this.subredditForm
      .get("subredditControl")
      .valueChanges.pipe(debounceTime(300),
distinctUntilChanged())
      .subscribe((subreddit: string) => {
        if (subreddit.length > 0) {
          this.redditService.reset(subreddit);
          this.contentArea.scrollToTop(200);
        }
      });
  }
```

First, we call the `getGifs` method in the Reddit service, and assign the result to our `gifs` member variable. This isn't related to our form at all, but we do need to trigger the initial loading of the data and this is a good place to do it.

We then retrieve the `subredditControl` form control we created inside of our `subredditForm` to get a reference to the observable that it provides. We `subscribe` to the observable so that every time it emits a value we run some code. This provides us with a convenient way to run some code every time the value changes, but we are taking things a little further than that. This is where we make use of those operators we imported from the RxJS library.

Instead of just using `valueChanges` to do something whenever the value changes, we chain on two operators using `pipe` to change the behaviour of the observable before we `subscribe` to it.

Basically, we can chain together as many of these operators as we want and they all modify the behaviour of the observable in some way. For example, if we just did this:

```
this.subredditControl.valueChanges.subscribe;
```

Then we would run some code every time the value of the `subredditControl` input changes. If we then changed it to this:

```
this.subredditForm.get("subredditControl").valueChanges.pipe(deboun
```

We would only run the code when the input has changed, and when there hasn't been another change within `0.3` seconds. This prevents us from sending out too many pointless requests to the API. If the user was typing `chemicalreactiongifs` for example, the code would be triggered for `c`, then `ch`, then `che`, then `chem` and so on until the full string has been typed. Not only will this send off a bunch of useless queries to the API, it's going to be a bad experience for the user as well as the list is constantly flickering and changing as they type. By adding debouncing, the code will only run once the full string has been typed in (assuming the user doesn't take more than `0.3` seconds between typing each letter).

Finally, we can add one final operator:

```
this.subredditForm
  .get("subredditControl")
  .valueChanges.pipe(debounceTime(1500),
distinctUntilChanged()).subscribe;
```

This will only run the code if the value is different to the last time it ran. So, if a user typed `gifs`, hit the backspace key to make it `gif` but then retyped the `s` to make it `gifs` again, nothing would happen. We don't need to reload the data for `gifs` because we are already there.

The end result is that the code will only trigger when the value has changed, the user is not currently typing and the input value is different to what it was last time. The code that we

are triggering simply checks if a non-empty value was supplied, and then changes the active subreddit by calling the `reset` method. We also call the `scrollToTop` method on the content area so that the list scroll back to the top automatically (otherwise, when the new gifs load in the user would have to manually scroll their way back to the beginning, which would be pretty annoying).

This will probably be one of the most confusing parts of this application, especially if you're completely new to observables. As you can tell this has allowed us to create some pretty useful functionality really easily, but we could have just as easily used the normal `ngModel` approach and just used a button for triggering searches instead. The downside to the `ngModel` approach is that we miss all of the powerful observable behaviour. Later in this application, we will make even heavier use of observable features.

Summary

We've got off to a really good start in this lesson, and we're well on the way to getting some cool stuff happening. We have a really nice basic structure which will allow us to easily build on functionality in the next lessons. If you take a look at the application now by running:

```
ionic serve
```

you should see something that looks like this:

GIF GOES HERE

TITLE GOES HERE

Pretty ugly right? Trust me, it'll all come together in the end! In the next lesson, we're going to look at pulling in some real data from Reddit.

NOTE: Remember, if you are running into any issues (e.g. errors in the console, functionality not working, or compilation errors in the terminal) make sure to stop serving (`CTRL + C`) and re-serve the application with `ionic serve`. Sometimes the live reloading fails to update the application properly, and you will need to re-serve the application.

Lesson 4: The Reddit API and HTML5 Video

One of the most interesting things about **Gif**list is that it won't ever contain any actual GIF files at all. The GIF file format can be quite large and slow to load, which is especially a problem on mobile when people may be more wary about using their data.

What we are going to do is make sure we only ever pull in gifs that are in the `.mp4` video format, or the **.webm** or **.gifv** format which can be used as an `.mp4` video. This means we won't be displaying GIFs at all, we will be displaying **videos**.

We're going to display those videos using the `<video>` tag. It's important to remember when building mobile applications with web tech, we can use modern web APIs within our applications. A great example of this is **Geolocation** - we can use the native API to access the devices GPS, but we can also just use the web-based Geolocation API available to any website. Basically, anything that you can do on a website you can do in your mobile application (but obviously we can do a lot more as well, given that we can also access native functionality).

Before we get into building this functionality, let's talk about some issues with the `<video>` tag.

HTML5 Video Behaviour on iOS and Android

Using a framework like Ionic means a lot of the platform differences are handled pretty seamlessly for us, but there's still going to be some problems with platform differences

that you will run into when creating cross-platform applications.

First, there are a few things you should know about the `<video>` element:

- It can be played either inline (i.e. the video just plays where it is without resizing) and in fullscreen mode
- It has a **poster** attribute that can be used to display an image before the video is loaded
- Whether controls are displayed, whether it auto plays video, whether it loops, and whether it plays inline can all be controlled

Exactly how the `<video>` element behaves will depend on the platform. Fortunately today, the behaviour of the `<video>` element is much more consistent than it was when I first wrote this book. It used to be that on some iOS devices even if you used the `inline` attribute it would still play in full screen on some devices. Now we can, more or less, just use the same video behaviour whether we are running on iOS, Android, or the web. However, there are some differences that need to be accounted for (we will deal with that a little later).

Fetching Data from Reddit

We're going to start off with the most interesting and complicated method in our Reddit service, which is pretty much the core feature of the entire application: `loadGifsFromActiveSubreddit()`. The idea is that this method will handle loading in gifs from whatever subreddit is currently set. We will add the code first and then walk through it.

> **Modify the `loadGifsFromActiveSubreddit` method in `src/app/services/reddit.service.ts` to reflect the following:**

```
  loadGifsFromActiveSubreddit(): void {
    //Build the URL that will be used to access the API based on
the users current preferences
    let url =
`https://www.reddit.com/r/${this.settings.subreddit}/${this.setting
limit=100`;

    // Only grab posts "after" this post if set (allows for
pagination)
    if (this.after) {
      url += `&after=${this.after}`;
    }

    this.loading = true;
    this.loading$.next(this.loading);

    this.http
      .get(url)
      .pipe(
        map((response: any) => {
          const posts: RedditPost[] = response.data.children;

          // Reddit can no longer return any data, stop searching
          if (posts.length === 0) {
            this.noPostsRemaining = true;
```

```
            return [];
          }

          // Map the response into the format we want to work with
          const modifiedPosts = posts.map((post) => {
            return {
              src: this.getBestSrcForGif(post),
              author: post.data.author,
              name: post.data.name,
              permalink: post.data.permalink,
              title: post.data.title,
              thumbnail: post.data.thumbnail,
              comments: post.data.num_comments,
              loading: false,
            };
          });

          // Filter out unuseable posts
          const filteredPosts = modifiedPosts.filter((post) => {
            return post.src !== false;
          });

          if (filteredPosts.length >= this.settings.perPage) {
            // Set the "after" variable to the page limit
            this.after = filteredPosts[this.settings.perPage -
1].name;

            // Only return enough posts to fill the page
            return filteredPosts.slice(0, this.settings.perPage);
```

```
          } else {
            // Set the "after" variable to the last post retrieved
(if one exists)
            if (modifiedPosts.length > 0) {
              this.after = modifiedPosts[modifiedPosts.length -
1].name;
            }

            // Return however many posts we were able to retrieve
            return filteredPosts;
          }
        })
      )
      .subscribe(
        (gifs: Gif[]) => {
          // Add new posts that were just pulled in to existing
posts
          this.gifs.push(...gifs);
          this.gifs$.next(this.gifs);

          if (this.timesRetried > 10) {
            // We've retried fetching gifs 10 times, time to give
up!
            console.log("Unable to fetch enough gifs: stopping");
            this.timesRetried = 0;
            this.loading = false;
            this.loading$.next(this.loading);
          } else {
```

```
            // Keep getting gifs until we have enough to fill a
page
            if (
              this.gifs.length < this.settings.perPage *
this.currentPage &&
              !this.noPostsRemaining &&
              !this.cancelSearch
            ) {
              this.timesRetried++;
              this.loadGifsFromActiveSubreddit();
            } else {
              this.loading = false;
              this.loading$.next(this.loading);
              this.timesRetried = 0;
            }
          }
        },
        (err) => {
          console.warn(err);
          this.loading = false;
          this.loading$.next(this.loading);
        }
      );
  }
```

That's a pretty big method. I've added some inline comments to help clear things up a little bit, but let's also talk about each bit of code in detail.

We start off by building the `url` that we will be making a `GET` request to. The Reddit JSON API is quite simple to use, all we need to do is slot in the subreddit we want to grab data from and the sort order into the URL, and then we can just make a `GET` request to it to return JSON data. If we have an `after` value set, we will also add that to the URL. This is used for pagination, if we supply an `after` parameter then the API will only return posts that come after the post that is used for the `after` value. That means if we grab posts `1-100` first, and then we use the `100th` post as the `after` value for the next `GET` request, the next time it will return us posts `101-200`. This is how we achieve pagination for this application.

Next, we set our `loading` flag to indicate that we are currently loading in gifs. We also call `next` on our `loading$` observable so that if anything is subscribed to it, it will be notified that a load has begun.

Then we make our `GET` request using the `HttpClient` to the `url` we just built. Instead of just subscribing to it for the result, which we could do, we first use `pipe` to supply a `map`. This `map` will modify the observable and return us a new observable instead. We can use this `map` to change the response in some way. What we are doing is modifying the response so that it only includes data in the format we want.

We first grab a reference to the actual posts through `response.data.children` which will match the `RedditPost` type we created earlier. Then we go through a series of steps to get the data in the format we need, which goes like this:

1. If the Reddit API isn't returning us any new posts, return immediately and set the `noPostsRemaining` flag
2. Use an array `map` (which is different to the observable `map`) to modify the `posts` to

only include the data we want

3. Use an array `filter` to remove any posts that we could not find a `src` for (e.g. no `.mp4` file was available)
4. Set the `after` value for the next call which will depend on whether we were able to find enough gifs to fill a page or not

With our data in the correct format, we are now able to `subscribe` to it. Rather than having to deal with messy data, most of which we won't even need, everything will now be nicely formatted. At this point we push the gifs we just retrieved into our `gifs` array, and call the `next` method on our `gifs$` observable.

This would be the end of the story, except that it is possible (and in some cases likely) that we were unable to return enough gifs to fill a page (`10`, `20`, or `30` gifs depending on what the user sets). So, we also need to check if we retrieved enough gifs and if not we will call the `loadGifsFromActiveSubreddit` method again. If you are not familiar with this already, this is called "recursion" (when a method calls itself). We could just let this keep running until it stops itself when it retrieves enough gifs, but it is also possible that it will *never* retrieve enough gifs, and we don't want an infinite loop. That means we also need to add checks to see if we have already made too many attempts, or if Reddit is no longer returning new posts. We also have a check for `cancelSearch` which indicates that the user is switching to another subreddit, and so we should cancel any active searches.

Finally, you should notice that we set the `loading` value appropriately once the search has finished. You might be relieved to have that method out of the way... but we're technically not done with it yet. You may notice that in our `map` we set the `src` property to this:

```
src: this.getBestSrcForGif(post),
```

We are making another method call to determine the "best source" for the gif. The response from the Reddit API is unfortunately quite messy. We want an `.mp4` file but sometimes we might have a `.gifv` or a `.webm` file. These files also aren't always in the same location inside of the response. Sometimes we will find the resource we want on the `url` property, sometimes it will be inside of the `media` property, sometimes it will be inside of the `secure_media` property, sometimes it will be in the `preview` property. Dealing with all these scenarios is a bit messy, which is why we have separated it out into its own method.

> Modify the `getBestSrcForGif` method in `src/app/services/reddit.service.ts` to reflect the following:

```
  getBestSrcForGif(post: RedditPost): string | boolean {
    // If the source is in .gifv or .webm formats, convert to .mp4
and return
    if (post.data.url.indexOf(".gifv") > -1) {
      return post.data.url.replace(".gifv", ".mp4");
    }

    if (post.data.url.indexOf(".webm") > -1) {
      return post.data.url.replace(".webm", ".mp4");
    }
```

```
    // If the URL is not .gifv or .webm, check if media or secure
media is available
    if (post.data.secure_media?.reddit_video) {
      return post.data.secure_media.reddit_video.fallback_url;
    }

    if (post.data.media?.reddit_video) {
      return post.data.media.reddit_video.fallback_url;
    }

    // If media objects are not available, check if a preview is
available
    if (post.data.preview?.reddit_video_preview) {
      return post.data.preview.reddit_video_preview.fallback_url;
    }

    // No useable formats available
    return false;
  }
```

The actual logic here isn't too bad. If the gif is available in the `.gifv` or `.webm` format we just use that and convert it to `.mp4`. Next, we try the `secure_media` property, then the `media` property, and finally the `preview` property. Note that the `secure_media`, `media`, and `preview` properties are not always included in the response from Reddit, so we use the safe navigation operator `?` to prevent errors if these properties don't exist. If none of these properties provide the `.mp4` we need, then we simply return `false` (which will then cause this result to be filtered out by our `filter` we just implemented).

Now that we have our list of gifs loading into the application, we can display real data in our list. But first, we will need to update our template to actually use the data.

> Modify the `<ion-list>` in `src/app/home/home.page.html` to reflect the following

```
<ion-list lines="none">
  <div class="gif" *ngFor="let gif of gifs">
    <ion-item button detail="false" (click)="playVideo($event,
gif)">
      <ion-spinner color="light" *ngIf="gif.loading"></ion-
spinner>
      <div
        style="background: url({{gif.thumbnail}}) 50% 50% / cover
no-repeat"
        [ngStyle]="gif.dataLoaded !== true ? {filter: 'blur(3px)
brightness(0.6)', transform: 'scale(1.1)'} : {}"
        class="preload-background"
      >
        <video
          playsinline
          poster="none"
          preload="none"
          [loop]="true"
          [muted]="true"
          [src]="gif.src"
        ></video>
```

```
      </div>
    </ion-item>
    <ion-list-header style="text-align: left">
      <ion-label> {{ gif.title }} </ion-label>
      <ion-button (click)="showComments(gif)"
        ><ion-icon name="chatbubbles"></ion-icon> {{ gif.comments
}}</ion-button
      >
    </ion-list-header>
  </div>
</ion-list>
```

We are now using an `*ngFor` to loop over every gif that our Reddit service has loaded, and we have added a `<video>` tag that uses the `src` of the `gif` to display the gif as a video. We've been busy with some other weird stuff in this snippet of code as well - let's talk through that.

We've added a `click` event to our `<ion-item>` that will call `playVideo`. We will implement the `playVideo` soon such that it takes the `$event` passed in to get a reference to the actual `<video>` element, and then call that videos `play` method (along with some other logic). Since we intend for this `<ion-item>` to be clicked, we give it the `button` attribute which will remove any tap delay (which can occur when adding `click` events to other elements that aren't intended to be interacted with like a `<div>` or an `<ion-item>` without the `button` attribute), and it is also better for accessibility.

We have added an `<ion-spinner>` that will display over the top of the video (once we

style it correctly) to indicate that the video is currently loading. The next addition we have made is probably the most complex addition:

```
<div
  style="background: url({{gif.thumbnail}}) 50% 50% / cover no-repeat"
  [ngStyle]="gif.dataLoaded !== true ? {filter: 'blur(3px) brightness(0.6)', transform: 'scale(1.1)'} : {}"
  class="preload-background"
></div>
```

Let's talk about the goal first. We won't be preloading video data for all of the videos on the page, so that means that by default the videos would just be blank rectangles - this isn't a great look for the user interface. Fortunately, the Reddit API also returns us a `thumbnail` of the video that we can use. We could just set this as the `poster` for the video, but it won't fill up all of the space the video occupies properly as it doesn't have the correct dimensions. Instead, we could use the `thumbnail` as a `background-image` for the `video` and resize it to fit. This can work okay, it will occupy the same space that the video occupies so we won't have weird spacing issues before the video is loaded in, but the downside is that the thumbnail is low resolution and will look low quality. This isn't a great look for our application either. Fortunately, we can make use of modern CSS to `blur` the thumbnail image. By going a bit over-the-top with the blurring we can make it look like the blur is intentional, and it looks like a mysterious preview of what is about to load in. This is a pretty good solution, yet there is still more trouble with it! If we `blur` the thumbnail image, it will cause the edges of the image to become fuzzy. It will look like

there is a strange kind of white "glow" to the edge of the image. This brings us to the final solution:

Instead of adding a background image to the `<video>` we instead add a surrounding `<div>` around the video and use the thumbnail as a background image for that. The trick here is to `scale` the `<div>` so that it is slightly larger than normal. This allows us to blur the image, but hide the ugly edges as they will be cropped out since the entire image can't fit in the container. This `blur` would also blur the video itself, so we need to make sure we only apply this style *before* the video has loaded. To achieve this, we use `ngStyle` to conditionally apply this style based on whether or no the video data for the gif has loaded yet.

Thought our trouble was over? Not quite. Eventually, we will need to style the `<video>` so that it has a `transparent` background, otherwise we won't be able to see the `preload-background` div behind the `<video>`. Unfortunately, on Android, if no `poster` is supplied the `<video>` doesn't just become transparent, it uses an (ugly) default poster for the video - so we won't be able to see our `preload-background` on Android. The final solution to this is to supply a `poster` to a non-existent image, which will fail to load in, and so we will be able to see what is behind the video on Android before the actual video loads in.

All of this effort was just for a relatively minor user interface improvement, but I think it's worth it. One other minor improvement we are going to make to our home page template is to add an `<ion-progress-bar>`. We are going to use this to indicate when gifs are currently in the process of being loaded in.

> Modify the `<ion-header>` in `src/app/home/home.page.html` to reflect the following

```
<ion-header>
  <ion-toolbar color="primary">
    <form [formGroup]="subredditForm">
      <ion-searchbar
        animated
        placeholder="subreddit..."
        formControlName="subredditControl"
        value=""
      ></ion-searchbar>
    </form>

    <ion-buttons slot="end">
      <ion-button (click)="openSettings()">
        <ion-icon slot="icon-only" name="settings"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
  <ion-progress-bar
    color="dark"
    *ngIf="redditService.loading"
    type="indeterminate"
    reversed="true"
  ></ion-progress-bar>
</ion-header>
```

We just display the `<ion-progress-bar>` whenever the `loading` flag in our Reddit

service is set to `true`.

Let's also add a bit of code to the `reset` method in our Reddit service so that the `loadGifsFromActiveSubreddit` function gets called when we run the application. We will be calling the `reset` method initially from our root component to kick off the gif fetching process, and after that, we can call the `reset` method whenever we want to switch to a new subreddit (or apply some new settings).

> Modify the `reset` method in `src/app/services/reddit.service.ts` to reflect the following:

```
  async reset(subreddit?: string): Promise<void> {
    this.cancelSearch = true;

    // Wait for any current fetching to finish
    this.getLoadingState()
      .pipe(takeWhile((loadingState) => loadingState === true,
true))
      .subscribe(async (loadingState) => {
        if (loadingState === false) {
          this.cancelSearch = false;
          this.currentPage = 1;
          this.gifs = [];
          this.noPostsRemaining = false;
          this.after = null;

          // TESTING: We will replace with values from settings
```

```
service later
          this.settings = {
            perPage: 20,
            subreddit: "gifs",
            sort: "hot",
          }

          if (subreddit) {
            this.settings.subreddit = subreddit;
          }

          this.loadGifsFromActiveSubreddit();
        }
      });
  }
```

Things can't just be easy right? In our `reset` method we want to reset all of our values that we use for searching like the current page, whether there are posts remaining, the current list of gifs, and so on. Once we have everything in a clean state, we set the `subreddit` (if one was passed in) and then call `loadGifsFromActiveSubreddit`. However, there is a danger to this method that we need to watch out for.

We can't just call `reset` whenever we want, because if the `loadGifsFromActiveSubreddit` method is already in the process of searching for gifs, and then we go and modify these values in the middle of a search, it is going to cause some problems. To work around this, we first wait for the `loading` state to change back to false. Rather than just subscribing to `getLoadingState` immediately, we first `pipe`

this `takeWhile` operator:

```
takeWhile((loadingState) => loadingState === true, true))
```

If you have been through the QuickLists walk-through then you will know that it is important for us to unsubscribe from subscriptions that we make (especially if we are going to be creating lots of subscriptions, as will be the case with this application). What this beautiful little operator will do is keep us subscribed until we receive a particular value back from the observable. In this case, we are saying that we want to keep taking values from this observable whilst the `loading` state is `true`. Once the loading state is no longer `true` it will automatically unsubscribe from the observable. An important thing to note here is that our code inside of the subscription checks to see if `loadingState` is `false` before executing the reset code. That means we need to receive **one** `false` value before terminating the description. Notice that our `takeWhile` operator looks like this:

```
takeWhile(/* condition */, true))
```

The second `true` parameter will make the `takeWhile` inclusive, meaning that when the `loadingState === true` check fails, it will still return us the first result to make it fail. This means that our `reset` code will run, and then the subscription will be terminated.

One last important thing we are doing here is setting the `cancelSearch` flag at the top of the method. If the `loadGifsFromActiveSubreddit` method is already performing a

search, it will automatically terminate once it sees that this flag has been set (which means we don't need to wait around for the existing search to entirely finish).

NOTE: Since we have not implemented the settings service yet, we are using a dummy value for `this.settings` just so that we can see some of this functionality working. We will replace that later with the actual values from the settings service.

Now we just need to call the `reset` method from somewhere to trigger our first search. We will call this in our root component so that the data starts loading as soon as the application is launched.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { RedditService } from "./services/reddit.service";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(private redditService: RedditService) {
    this.initializeApp();
```

```
  }

  async initializeApp() {
    this.redditService.reset();

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#e74c3c" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

If you load up your application in the browser now, you should see something like this:

subreddit...

kids on a playground ride 2518

Natures bottle opener 378

It's still not looking great, but it's looking a lot better and we have some cool gifs displaying in there now (which unfortunately we can't actually play just yet). Let's change that.

Playing our GIFs (videos)

Our gifs are sitting in our list now ready to go, we just need to get them movin' and shakin'. To do that, we are going to have to finish implementing the `playVideo` method.

> Modify the `playVideo` method in `src/app/home/home.page.ts` to reflect the following:

```
  playVideo(ev: Event, gif: Gif): void {
    const video = ev.target as HTMLVideoElement;

    if (video.readyState === 4) {
      if (video.paused) {
        video.play();
      } else {
        video.pause();
      }
    } else {
      if (video.getAttribute("data-event-loadeddata") !== "true")
{
        gif.loading = true;
        video.load();
```

```
        const handleVideoLoaded = async () => {
          await video.play();
          gif.dataLoaded = true;
          gif.loading = false;
          video.removeEventListener("loadeddata",
handleVideoLoaded);
        };

        video.addEventListener("loadeddata", handleVideoLoaded);
        video.setAttribute("data-event-loadeddata", "true");
      }
    }
  }
```

This is another one of those methods that seems a little bit more complicated than you might think it would be, because we're handling all of these little edge cases to make the experience nicer. At a high level we are doing this when the user taps the video:

- If the video is loaded and not currently playing, `play` it
- If the video is loaded and already playing, `pause` it
- If the video has not been loaded yet, `load` it, and then `play` it

First of all, we have a reference to the `<video>` element itself which we are able to grab from the `target` of the click event. We can't give the `video` reference a type of `HTMLVideoElement` directly (which gives us access to autocompletion for the videos methods and properties), because an event target that we get from a click event could be

a range of different things. Instead, we treat `ev.target` as a `HTMLVideoElement` by using the `as` keyword. This is something you don't need to do often, but comes in handy when you do end up needing it.

We check the `readyState` and a `readyState` of `4` means that enough data for the video has loaded such that it can play all the way through. We have the `preload` for our `<video>` elements set to `none` so none of the videos will be in a `readyState` of `4` when they are first clicked.

What we do if the video element does not have a `readyState` of `4` is we call the `load` method. Since we also want to set the `loading` and `dataLoaded` properties of our gif appropriately once the video is finished loading (which we are making use of in our template, e.g. we want to only display the loading indicator whilst the video is loading) we need to set up an event listener for `loadeddata`. The `loadeddata` event will be triggered once the data has finished loading. When that happens, we call our `handleVideoLoaded` function, at which point we also remove the listener.

There is one more situation we are dealing with here. We only want to add the `loadeddata` event listener once, but people are impatient, and you can imagine people might tap the video multiple times whilst waiting for it to load. We don't want to add multiple event listeners for the video when this happens, we just want one. To deal with this, we set a custom data attribute called `data-event-loadeddata` to `true` when we add the listener (I just made this attribute up, it doesn't have any special meaning, you could use `data-anything-you-like`). Now if this method is called again, we first check to see if this `data-event-loadeddata` attribute has been already set before creating the event listener.

Launching Comments in the InAppBrowser

Remember earlier when we created a reference to the `Browser` API that Capacitor provides? Well, we are going to make use of that now. When the user clicks on the comments button on the video we want to launch a browser with the link to the original submission of the post.

> Modify the `showComments` method in `src/app/home/home.page.ts` to reflect the following:

```
showComments(gif: Gif): void {
  Browser.open({
    toolbarColor: "#fff",
    url: `https://reddit.com${gif.permalink}`,
    windowName: "_system",
  });
}
```

Compared to the others, this is a pretty simple method. We simply grab the link from the gifs data and use `Browser` to launch it by calling the `open` method that it provides.

Loading More GIFS

One important method we haven't go to yet is the `loadMore` method that is called by our `<ion-infinite-scroll>` component. What we want to do when this happens is load in

the next page of gifs beneath what is currently already in the list.

> Modify the `loadMore` method in `src/app/home/home.page.ts` to reflect the following:

```
  loadMore(ev: Event): void {
    const infiniteElement = ev.target as
HTMLIonInfiniteScrollElement;

    if (!this.redditService.loading) {
      this.redditService
        .getLoadingState()
        .pipe(
          skip(1),
          takeWhile((loadingState) => loadingState === true, true)
        )
        .subscribe((loadingState) => {
          if (loadingState === false) {
            infiniteElement.complete();
          }
        });

      this.redditService.nextPage();
    } else {
      infiniteElement.complete();
    }
  }
```

Again, we use the `target` of a click event to get a reference to our infinite scroll element. This is important as once we have finished loading in the data we want, we need to call its `complete` method to indicate that the loading is complete.

First, we check if the Reddit service is already loading and if it is we just immediately complete the infinite scroll and we don't load any new data. Otherwise, we make a call to the `nextPage` method in our Reddit service to load in the next page of data.

However! Once again, it isn't quite that easy. We need to call the `complete` method once the new data has finished loading in. We **can not** just do this:

```
this.redditService.nextPage();
infiniteElement.complete();
```

Because the `complete` will be triggered immediately after the call to `nextPage`. However, `nextPage` will trigger the `loadGifsFromActiveSubreddit` method which will be executed asynchronously and might take a while to actually complete. We could set up our `nextPage` method such that it returned a promise that we could `await` or an observable we could subscribe to, but we already have a simple way to detect when loading is completed through our `loading$` observable that `getLoadingState()` returns.

What we do is subscribe to the loading state, and once the `loadingState` returns as `false` (i.e. it is no longer loading) we trigger the `complete` method. We use our `takeWhile` operator again here to automatically unsubscribe after we receive that `false`

value. Also notice that we use a new operator called `skip`. This will skip the first value emitted by the observable. We want to do this because we are checking if `loadingState === false` before we call the `complete` method, but the first value will always be `false` because the loading has not begun yet. For example, if our loading observable outputs these values:

- false
- true
- true
- false

We want to wait for that second `false` value which indicates that the loading has begun and then finished. The first `false` value just indicates that the loading hasn't begun yet, so we `skip` that one.

Now let's just quickly finish up the `nextPage` method in our Reddit service.

> Modify the `nextPage` method in `src/app/services/reddit.service.ts` to reflect the following:

```
nextPage(): void {
  this.currentPage++;
  this.loadGifsFromActiveSubreddit();
}
```

Thankfully, all we need to do for this method is increment the `currentPage` value and call the `loadGifsFromActiveSubreddit` method.

Summary

We've completed a big chunk of the application in this lesson. You should be able to use `ionic serve` to serve the application in the browser now, and you should be able to see some stuff happening. However, there are still a few loose ends that we need to tie up before everything is working properly.

Lesson 5: Integrating the Settings Service

In this lesson, we are going to finish up the services that our application provides. First, we are going to implement our settings service so that we can save and load user preferences in the application.

Then, we are going to finish implementing the Reddit by updating it to make use of the settings service (rather than just using a dummy settings object). We've already done most of the heavy lifting so there isn't actually much to do here.

The Settings Service

> Modify `src/app/services/settings.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { Storage } from "@ionic/storage";
import { Settings } from "../interfaces/settings";

@Injectable({
  providedIn: "root",
})
export class SettingsService {
  constructor(private storage: Storage) {}

  async getSettings(): Promise<Settings> {
```

```
    let settings = await this.storage.get("settings");

    if (settings == null) {
      settings = {
        perPage: 20,
        subreddit: "gifs",
        sort: "hot",
      };
    }

    return settings;
  }

  save(settings: Settings) {
    this.storage.set("settings", settings);
  }
}
```

All we need to do here is save some data that is passed in to our `save` method using the Ionic's Storage API on the `settings` key, and we also need to be able to retrieve that data. Notice that we use our `Settings` interface to ensure that the data that is passed in is in the correct format. We also add a check when retrieving the data to make sure that something under they `key` of `settings` exists actually exists. If this is the first time the user has used the application or otherwise they haven't set any settings yet, there won't be anything in storage. If there are no settings in storage, then we just return a default settings object.

Finishing the Reddit Service

With the settings service implemented (as simple as it is) we are now going to make use of that in our Reddit service.

> Modify the `reset` method in `src/app/services/reddit.service.ts` to reflect the following:

```
  async reset(subreddit?: string): Promise<void> {
    this.cancelSearch = true;

    // Wait for any current fetching to finish
    this.getLoadingState()
      .pipe(takeWhile((loadingState) => loadingState === true,
true))
      .subscribe(async (loadingState) => {
        if (loadingState === false) {
          this.cancelSearch = false;
          this.currentPage = 1;
          this.gifs = [];
          this.noPostsRemaining = false;
          this.after = null;
          this.settings = await
this.settingsService.getSettings();

          if (subreddit) {
            this.settings.subreddit = subreddit;
```

```
        }

        this.loadGifsFromActiveSubreddit();
      }
    });
  }
```

All we have changed here is the way that `this.settings` is assigned. Now we are awaiting the promise returned by `this.settingsService.getSettings()` which will resolve with our settings object.

With these finishing touches on our services, our application should **mostly** be working completely now. The most noticeable thing that is missing is the styling, so you won't have a particular good experience watching the videos because the sizing and positioning will be wrong - but it should still work. We also still need to implement the settings page so that the user can modify and save the settings using the settings service we just implemented.

Lesson 6: The Settings Page

We have everything working pretty nicely now - there are GIFs coming in from Reddit, filtered to only include the file format we want and we have them displayed using `<video>` elements (which we still need to style properly).

We currently have the **gifs** subreddit set as the default, which is certainly a good choice for this app, and the user can change this if they like. When they leave the application and come back, though, it is going to get set back to the default **gifs** subreddit. This would get pretty annoying if they aren't interested in looking at the **gifs** subreddit.

So, what we are going to do is create a **Settings** page, where the user can set the following:

- Default subreddit
- GIFs to load per page
- Sort order

Creating the Settings Page

To open the settings page we are going to use a **Modal**. Opening a page as a Modal is a bit different to just opening a page normally in that it is completely separate to the navigation flow of the app. A Modal is basically a single page that can be opened over the top of the current content and then closed (I'm sure you're familiar with other modal windows around the web like Facebook's photo viewer).

Unlike our other pages where we have a route set up to activate them, for our modal we just supply the page component directly to the `ModalController`. Before we worry about that, let's create the template for the settings page first.

> Modify `src/app/settings/settings.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title> Settings </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="close()"><ion-icon slot="icon-only"
name="close"></ion-icon></ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-card>
    <ion-card-content>
      <p>
        <strong>Giflist</strong> is a client for
<strong>reddit</strong> that will
        <strong>stream GIFs</strong> from <strong>any subreddit
that contains gifs</strong>. By
        default it uses the popular <strong>gifs</strong>
subreddit, but you can change this to
        whatever you like, e.g: <em>perfectloops</em>,
```

```
<em>chemicalreactiongifs</em>,
        <em>animalsbeingderps</em>.
      </p>
      <br />
      <p>
        To play a GIF, just tap it. To view the original
submission for any GIF, tap the title
        section and it will open in reddit. You can configure some
settings for the application
        below. Enjoy!
      </p>
    </ion-card-content>
  </ion-card>

  <h3>Default Subreddit</h3>

  <ion-input
    clearInput
    type="text"
    placeholder="subreddit..."
    [(ngModel)]="settings.subreddit"
  ></ion-input>

  <h3>Sort</h3>

  <ion-segment color="primary" [(ngModel)]="settings.sort">
    <ion-segment-button value="hot"> Hot </ion-segment-button>
    <ion-segment-button value="new"> New </ion-segment-button>
  </ion-segment>
```

```
  <h3>Page Size</h3>

  <ion-segment color="primary" [(ngModel)]="settings.perPage">
    <ion-segment-button value="10"> 10 </ion-segment-button>
    <ion-segment-button value="20"> 20 </ion-segment-button>
    <ion-segment-button value="30"> 30 </ion-segment-button>
  </ion-segment>
</ion-content>

<ion-footer>
  <ion-button expand="full" (click)="save()" color="primary">Apply
Settings</ion-button>
</ion-footer>
```

There's nothing too different to what we have already been doing going on in this template. We define our header as usual, and we add a button to it which will call a `close` method we will define shortly to dismiss the modal window.

We add an `<ion-card>` to display some information about the app to the user, and then we define a few different inputs. We use a simple text input for setting the subreddit, but we use segment components to control the **sort** and **perPage** values. All of these inputs have two way data binding set up with `[(ngModel)]` so we will easily be able to access these values in our class definition, which we will create now.

> Modify `src/app/settings/settings.page.ts` to reflect the following:

```
import { Component, OnInit } from "@angular/core";
import { ModalController } from "@ionic/angular";
import { SettingsService } from "../services/settings.service";
import { Settings } from "../interfaces/settings";

@Component({
  selector: "app-settings",
  templateUrl: "./settings.page.html",
  styleUrls: ["./settings.page.scss"],
})
export class SettingsPage implements OnInit {
  public settings: Settings = {
    perPage: 20,
    subreddit: "gifs",
    sort: "hot",
  };

  constructor(private modalCtrl: ModalController, private
settingsService: SettingsService) {}

  async ngOnInit() {
    this.settings = await this.settingsService.getSettings();
  }

  save() {
    this.settingsService.save(this.settings);
    this.close();
  }
```

```
  close() {
    this.modalCtrl.dismiss();
  }
}
```

When this page is initialised, we make a call to our settings service to get whatever settings are currently in storage (or the default settings object if there are no settings in storage). We then use this as our settings object, which will fill out the form inputs on the page with whatever the current values are.

When the user hits the save button on the page we call the `save` method, which will pass our current settings object through to the `save` method of the settings service and then close the modal.

An interesting thing about modals is that since they behave a little differently to the normal flow of navigation in our application, the way in which we navigate away from them is a little different. We have implemented a `close` method here that calls the `dismiss` method of the `ModalController` which will dismiss the modal and go back to the page we were on previously. The user can either trigger this directly by clicking the close button in the template, or by saving their settings in which case it is called automatically.

Opening the Settings Page as a Modal

Now that we have our Settings page defined, we just need to launch it as a Modal. We've already set up a button in the header of our home page to open the settings page, so all we need to do is finish defining the `openSettings` method.

> Modify the `openSettings` method in `src/app/home/home.page.ts`:

```
async openSettings(): Promise<void> {
  const modal = await this.modalCtrl.create({
    component: SettingsPage,
  });

  modal.present();

  await modal.onDidDismiss();
  this.redditService.reset();
}
```

As you can see above, we pass the `SettingsPage` we created (which is being imported into the home page) into the `create` method of the `ModalController`. This method returns a promise, and when that promise resolves we can use the result to display the modal on the screen by calling `modal.present()`.

We also wait for `modal.onDidDismiss()` to resolve, which will occur once the modal has been closed/dismissed. When this happens, we make a call to our `reset` method in the Reddit service to apply any new settings that were supplied by re-fetching data with the new settings.

Summary

With the settings page implemented, the functionality of our application is completely

finished! It's still pretty ugly though...

So, we have one lesson remaining where we are going to add a few styles to make everything look a lot nicer.

Lesson 7: Styling

We're almost there, just a little bit of styling to go and we'll have a completed application.

> Modify `src/app/home/home.page.scss` to reflect the following

```
ion-searchbar {
  padding: 0 5px;
}

ion-list {
  padding: 0;
}

ion-buttons {
  margin: auto 0;
}

ion-label {
  margin: 0;
  padding: 10px 0;
  overflow: auto;
}

.gif ion-item {
  --inner-padding-end: 0;
```

```
  --padding-start: 0;
  position: relative;
}

.gif ion-spinner {
  margin: auto;
  position: absolute;
  left: 0px;
  right: 0px;
  z-index: 1;
  background-color: var(--ion-color-dark);
  border: 10px solid var(--ion-color-dark);
  border-radius: 5px;
  padding: 20px;
}

.comments {
  display: block;
  width: 100%;
  margin-top: 5px;
  text-align: right;
  color: var(--ion-color-medium);
}

ion-list-header {
  align-items: center;
  background-color: var(--ion-color-light);
  border-bottom: 10px solid var(--ion-color-medium);
```

```
}

ion-list-header ion-button {
  margin: 0;
}

.preload-background {
  width: 100%;
  height: auto;
}

video {
  width: 100%;
  height: auto;
  margin: auto;
  background: transparent;
}

ion-infinite-scroll-content {
  margin-top: 20px;
}
```

These are all pretty basic styles, for the most part, we are just changing some colours and messing with padding and margins. The most important styles here are the `video` styles which serve to make sure the video scales and sits nicely in the list. We make sure the video always fills up the full width of the screen, and automatically scales its height so that it uses the correct resolution. Another important style here is the style for `ion-spinner`. This is displayed when the video is loading, and we want it to overlay the video. This is

why we give it a higher `z-index` and we use an `absolute` position to center it. Since its parent `ion-item` has a `position` of `relative`, the `absolute` positioning on the `ion-spinner` will cause it to be positioned relative to the `ion-item` (not the entire screen which is usually the case for `absolute` positioning).

We are also going to add a few styles to our settings page.

> Modify `src/app/settings/settings.page.scss` to reflect the following

```
ion-card,
ion-segment {
  --ion-background-color: #fff;
}

ion-input {
  background-color: #fff;
  font-size: 1.2em;
  --padding-start: 15px;
  --padding-end: 15px;
  --padding-top: 15px;
  --padding-bottom: 15px;
}
```

Again, just some minor style tweaks here. Notice that we are using CSS variables to change the padding as the `ion-input` component makes use of these variables.

> Add the following style to the bottom of `src/global.scss`:

```
ion-content {
  --ion-background-color: var(--ion-color-light);
}
```

We are setting the `--ion-background-color` to our light theme colour here as we want most of our components to use a light colour for their background. Note that we do override this in some cases (e.g. on the settings page we manually supply an `--ion-background-color` of `#fff` for the `ion-card` and `ion-segment` which will override the value supplied in `global.scss`).

Finally, we are going to modify the CSS variables for our theme. I have already created a theme for us to use, but a great way to generate your own themes is to use the Ionic [Color Generator]() tool.

> Modify the `:root` selector in `src/theme/variables.scss` to reflect the following:

```
:root {
  --ion-color-primary: #e74c3c;
  --ion-color-primary-rgb: 231, 76, 60;
  --ion-color-primary-contrast: #ffffff;
  --ion-color-primary-contrast-rgb: 255, 255, 255;
  --ion-color-primary-shade: #cb4335;
```

```
  --ion-color-primary-tint: #e95e50;

  --ion-color-secondary: #ecf0f1;
  --ion-color-secondary-rgb: 236, 240, 241;
  --ion-color-secondary-contrast: #000000;
  --ion-color-secondary-contrast-rgb: 0, 0, 0;
  --ion-color-secondary-shade: #d0d3d4;
  --ion-color-secondary-tint: #eef2f2;

  --ion-color-tertiary: #5260ff;
  --ion-color-tertiary-rgb: 82, 96, 255;
  --ion-color-tertiary-contrast: #ffffff;
  --ion-color-tertiary-contrast-rgb: 255, 255, 255;
  --ion-color-tertiary-shade: #4854e0;
  --ion-color-tertiary-tint: #6370ff;

  --ion-color-success: #2dd36f;
  --ion-color-success-rgb: 45, 211, 111;
  --ion-color-success-contrast: #000000;
  --ion-color-success-contrast-rgb: 0, 0, 0;
  --ion-color-success-shade: #28ba62;
  --ion-color-success-tint: #42d77d;

  --ion-color-warning: #ffc409;
  --ion-color-warning-rgb: 255, 196, 9;
  --ion-color-warning-contrast: #000000;
  --ion-color-warning-contrast-rgb: 0, 0, 0;
  --ion-color-warning-shade: #e0ac08;
  --ion-color-warning-tint: #ffca22;
```

```
  --ion-color-danger: #eb445a;
  --ion-color-danger-rgb: 235, 68, 90;
  --ion-color-danger-contrast: #ffffff;
  --ion-color-danger-contrast-rgb: 255, 255, 255;
  --ion-color-danger-shade: #cf3c4f;
  --ion-color-danger-tint: #ed576b;

  --ion-color-dark: #222428;
  --ion-color-dark-rgb: 34, 36, 40;
  --ion-color-dark-contrast: #ffffff;
  --ion-color-dark-contrast-rgb: 255, 255, 255;
  --ion-color-dark-shade: #1e2023;
  --ion-color-dark-tint: #383a3e;

  --ion-color-medium: #c5c8d4;
  --ion-color-medium-rgb: 197, 200, 212;
  --ion-color-medium-contrast: #000000;
  --ion-color-medium-contrast-rgb: 0, 0, 0;
  --ion-color-medium-shade: #adb0bb;
  --ion-color-medium-tint: #cbced8;

  --ion-color-light: #f4f5f8;
  --ion-color-light-rgb: 244, 245, 248;
  --ion-color-light-contrast: #000000;
  --ion-color-light-contrast-rgb: 0, 0, 0;
  --ion-color-light-shade: #d7d8da;
  --ion-color-light-tint: #f5f6f9;
}
```

Ionic also comes with styling for a **dark mode** by default, which will be enabled if the users preferences are set to use dark mode where available. We will be removing that, as it will mess with the styles that we have set up above.

> Remove everything inside of `@media (prefers-color-scheme: dark) { }` in `src/theme/variables.scss`:

Alternatively, you could supply your own different set of styles that you want to be activated when dark mode is switched on. I have this as a homework exercise in the next lesson, so if you would like to have a go at implementing your own dark mode theme you should keep the current contents of `@media (prefers-color-scheme: dark) { }` so that you have a base to work from.

If you load up the application should now look something like this:

subreddit...
Money well spent?
63
Animation showing the location of "superplumes" around the Earth's core
51

Much better! If you take a look at the Android version (you can do this by setting the Emulator in Chrome Dev Tools to emulate a Samsung or other Android device or just use the `ionic serve -l` command) you will find it looks like this:

It certainly looks a little different, but it still looks good. One of the best things about Ionic is that it automatically conforms to design conventions of the platform it is running on as best it can. So in general, the less you mess with this the better. If you mainly use an iOS device, or if you mainly use an Android device, the styling of the other platform may look a little weird to you, but it won't look weird to users of that platform. Unless you have good reason to, you shouldn't modify the styling of one platform to look like the other (which is

certainly possible).

Summary

This was a pretty short lesson compared to the others, but it shows how easy it can be with a little effort to create a nice custom theme for your app.

Conclusion

Congratulations on making it through the Giflist application. We've learned a lot through developing this application, but the main points are:

- How to fetch data from a 3rd party API
- How to create complex interfaces
- How to manipulate data
- How to store and retrieve data locally
- How to use and manipulate observables
- How to use conditionals in templates
- How to make use of events
- How to display and interact with videos
- How to integrate Capacitor functionality

There's always room to take things further though, especially when you're trying to learn something. Following tutorials is great, but it's even better when you figure something out for yourself. Hopefully, you have enough background knowledge now to start trying to extend the functionality of the application by yourself, here are a few ideas to try out:

- Create your own custom theme (or dark mode theme) for the application **[EASY]**
- Create a **Random** option that will display gifs from a random subreddit (chosen from a predefined array) **[EASY]**
- Modify the application to display only still images instead of gifs **[MEDIUM]**
- Modify the application to display both still images and gifs **[HARD]**
- Allow users to save their favourite gifs into its own list that can be viewed by typing

`favorites` in the search bar **[VERY HARD]**

Remember, the Ionic documentation is your best friend when trying to figure things out.

What next?

You have a completed application now, but that's not the end of the story. You also need to get it running on a real device and submitted to app stores, which is no easy task. The final sections in this book will walk through how to take what you have done here, and get it published. If this is something that you are interested in learning now, make sure to check those sections out.

Snapaday

Lesson 1: Snapaday Introduction

Snapaday is based around one of the most commonly used device features: the camera. I don't actually have any data to back that claim up, but the camera is definitely one of the more common integrations. It's also a plugin people often struggle with, it's one of those things that's easy enough to use at a basic level, but there are a few things you need to be aware of to properly retrieve, store, and display photos.

Snapaday is sort of the "native plugin app" in this book - as well as covering how to use the camera we'll also be integrating local notifications and social sharing (we've got to get those selfies on Facebook right?) with additional Capacitor plugins. When it comes to native functionality integration, the line between what is possible on the web and what is strictly native only is a bit blurry. Some plugins (even the Camera plugin) *can* work directly in the browser if you choose to build your application as a PWA, but some don't. This is one of the main goals of a tool like Capacitor - where possible it will allow for the use of web implementations of the same native functionality that you are integrating.

We will be using this application as an example of how we can get Capacitor plugins working on the web. By using the `@ionic/pwa-elements` package we will be able to get our Camera functionality working on the web as well as on iOS and Android. The `@ionic/pwa-elements` package basically adds support for common native interfaces like the Camera dialog for applications that are running as a PWA - keep in mind that not all Capacitor plugins have a web implementation, and that many will only function when

your application is running natively on iOS or Android.

There are some minor modifications we will need to make to get the web functionality working, but we can more or less use the same code and have it work across all three platforms. Keep in mind that we will be storing the photos taken with this application in native storage on iOS and Android, the web version of this application will just be for the sake of demonstration and testing as there is not a good local storage option for photos on the web (if you wanted to build this application as a PWA you would likely want to have the photos uploading to some kind of cloud storage).

About Snapaday

The general idea for this application is that the user takes a photo with the application every day, and then they can play back their photos in a fast slideshow to see how they've changed over time (ever see this video?). To be more precise, the exact features of the application will include:

- Allowing the user to take a photo (only once per day)
- Displaying a list of all photos the user has taken in a list
- Allowing the user to delete any photos they no longer want
- The ability to play photos back in a fast slideshow format
- The ability to share photos they have taken
- Daily reminders with local notifications

Throughout building this some concepts you will learn are:

- How to integrate native plugins
- How to use the Camera API

- How to use the File API
- How to use local notifications
- How to use modals
- How to create a custom service
- How to store photos permanently
- How to use the `@ionic/pwa-elements` package

and here's a look at what we'll be building:

snapaday
It's time to take your selfie!
(tap here)

1 days ago

Lesson Structure

1. Getting Ready
2. The Layout
3. Taking Photos with the Camera
4. Storing and Retrieving Photos
5. Creating a Custom Pipe and Flipbook of all Photos
6. Integrating Local Notifications
7. Styling

Ready?

Now that you know what you're in for, let's get to building it!

Lesson 2: Snapaday Getting Ready

In this lesson, we are going to prepare our application for the journey ahead. We are going to generate the application, and we are also going to set up all of the components, plugins, and routes that we need. The idea is to get all of the general scaffolding required for most projects out of the way so we don't have to mess around with creating files and configuring things throughout the rest of the lessons.

At the end of this first lesson, we should have a nice skeleton application set up with everything we need to start diving into coding.

A good rule of thumb before starting any new application is to make sure you have the latest version of the Ionic CLI installed, so if you haven't done it recently then make sure to run:

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

If your environment is not set up correctly, you may run into permission errors when installing this package globally. If you are having permission issues, you can use the Ionic teams guide to solving these issues: Resolving permission errors.

If you do not already have a recent version of Node/npm set up on your machine, make sure to complete the Environment Setup guide first.

Generate a new application

We will be using the blank starter template for this application which, as the name implies, is basically an empty Ionic project. It comes with one page built in called **HomePage** which we will reuse as our main page that will hold our feed of photos, and we will also create an additional **SlideshowPage** to use for our slideshow modal.

> Run the following command to generate a new application

```
ionic start snapaday blank --type angular
```

When you are prompted with the following:

```
? Integrate your new app with Capacitor to target native iOS and
Android? (y/N)
```

Say **Yes**, as we will be utilising Capacitor for native builds. If you do not set up Capacitor when first creating your application, you can always do it manually later as well by running the following command:

```
ionic integrations enable capacitor
```

NOTE: You should still install Capacitor in this project even if you do not intend to build for iOS or Android, as we will still be using Capacitor for the web platform as well.

> Make the new project your current working directory by running the following

command:

```
cd snapaday
```

Your project should now be generated and you can open up the project folder in your favourite editor. You can take a look at the application as it is now by running the following command:

```
ionic serve
```

which for now should look something like this:

Blank

Ready to create an app?

Start with Ionic UI Components

Create the Required Components

The structure of this application is pretty simple, all we're going to have is two pages: the home page which will contain the photos and a page for the slideshow. We've already got a home page component which we will use as our photo list page, so we only need to create the **Slideshow** page.

NOTE: You will need to stop serving your application in order to run these commands. If your application is currently being served through the command line, make sure to hit `Ctrl + C` first to stop the process.

> Run the following command to generate the Slideshow page:

```
ionic g page Slideshow
```

Create the Required Services

As well as our extra page, we are also going to create a couple of services. We will create a photo service to handle all of the operations related to taking and storing photos, and since we'll be using a few alerts in the application we will be creating a service to make creating those alerts using Ionic's `AlertController` a little easier.

> Run the following command to generate a Photo service:

```
ionic g service services/Photo
```

NOTE: Notice that we use `services/Photo` instead of just `Photo` this time. This is because we want to generate all of our services inside of a folder called `services`.

> Run the following command to generate an SimpleAlert service:

```
ionic g service services/SimpleAlert
```

Create a Pipe

We will also be creating a **Pipe** for this application. A pipe is an easy way for us to manipulate data before it is displayed to the user in the template. Pipes are typically used to do things like display currency in a user friendly format. We will be using a pipe to convert our date formats into something like `2 days ago` when it is displayed to the user.

> Run the following command to generate a DaysAgo pipe:

```
ionic g pipe pipes/DaysAgo
```

Create the Interface

We will eventually be defining a custom interface to represent our `Photo` type. There is no generate command for this, so you will need to create it manually.

> Create a new folder inside of the `app` folder called `interfaces`

> Create a new file inside of the interfaces folder called `photo.ts`

Configure the Routes

As we discussed in the basics section, we need to define routes so that our application can match up the current URL to a particular component that you want to display. Generating the application and using the generate commands do most of the work for us, as routes are automatically injected into the **app-routing.module.ts** file. However, we will need to make some slight modifications to these.

> Modify `src/app/app-routing.module.ts` to reflect the following:

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

const routes: Routes = [
  {
    path: "home",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
  },
  {
    path: "",
    redirectTo: "home",
    pathMatch: "full",
  },
];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Notice that we do not have a route for the **Slideshow** page? This is because we will not be using the `<ion-router-outlet>` to display our slideshow page - we will be launching it as a modal inside of our home page.

IMPORTANT: You will need to delete the **src/app/slideshow/slideshow.module.ts** file, otherwise, you will get errors when building your application. Since we are using the component directly as a modal, it does not need a module file.

Configure Capacitor

At this point, you should already have Capacitor added to your project. However, if you did not respond **yes** to the prompt about installing Capacitor when initially creating the project, you should manually set it up now.

> Run the following command (if required) to enable Capacitor:

```
ionic integrations enable capacitor
```

Keep in mind that by default your application will be given a **Bundle ID** of `io.ionic.starter`. This Bundle ID is used to identify your native iOS/Android builds, and you will eventually want to change this to something unique to you or your company. You can change it later if you wish, but it is easier to do before we add the iOS and Android platforms. You may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`. If you don't intend on eventually modifying and publishing this project, then this step doesn't matter.

If you look at the **capacitor.config.json** file, you will notice that the `webDir` is listed as `www`. This is the folder that Capacitor will look to when it is copying over the code for your application into the native projects. It doesn't care about the rest of your source code, it only cares about the built output (i.e. the code that is actually run through the browser). By default, the `www` folder will not exist in your project until you perform a build of your application manually. We are going to do that now.

> Run the following command to generate a build of your application:

```
ionic build
```

It is good to use the standard `ionic build` command whenever you want to test your application on a device during development, but keep in mind that this creates a development build. When you are building the final version of your application, or if you want to test a production version of your application, you should run:

```
ionic build --prod
```

which will create an optimised production build.

Add Native Platforms

By default, the iOS and Android platforms will not be enabled in your project. If you want to add these platforms you can do so now (or you can do it later if you prefer). Remember, building for iOS will require Xcode and building for Android will require Android Studio - if you want to add these platforms now, you should make sure you have followed the instructions in the Generating an Ionic Application lesson to install Xcode, Android Studio, and their dependencies.

```
ionic cap add ios
```

Keep in mind that the first time the `Updating iOS native dependencies` line is run it may take quite a while.

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your Ionic application

code and dependencies at any time by running:

```
ionic cap sync
```

IMPORTANT: If you want to use a production build with Capacitor, you will need to add the `--prod` flag to the sync command, e.g: `ionic cap sync --prod`.

When you make changes to your application build, you will need to run this command to copy the changes over to the native iOS/Android projects created by Capacitor. If you do not run this command, any changes you make to your Ionic application will not be present when running the application on iOS/Android.

Set up Root Component

The default starter templates for Ionic utilise **Ionic Native** to handle hiding the **Splash Screen** and styling the **Status Bar**. Whilst you can still use **Ionic Native** and **Cordova** plugins inside of a Capacitor project, Capacitor provides this functionality through default APIs, so we are going to use those instead. It is also important to remember that the Cordova Splash Screen plugin is not compatible with Capacitor, so make sure that you do not install it in your project.

> Modify `src/app/app.component.ts` to reflect the following:

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

```
import { Plugins } from "@capacitor/core";

const { SplashScreen, StatusBar, LocalNotifications } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor() {
    this.initializeApp();
  }

  async initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#eb445a" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

You can also use the `StatusBar` API to hide the status bar entirely, but in this case we are styling it with a colour that will match our application. The status bar is the bar at the

top of native applications that display the time, battery level, etc.

Notice that we are also importing the `LocalNotifications` plugin form Capacitor's `Plugins`. We will be making use of this later.

Set up PWA Elements

I mentioned in the beginning that we will be using `@ionic/pwa-elements` with this application so that we can get the `Camera` API working even when we are just running the application through the browser/on the web. Without PWA Elements, the `Camera` API will only work when deployed natively to iOS/Android.

> Install PWA Elements with the following command:

```
npm install @ionic/pwa-elements
```

In order to use PWA Elements, we will also need to add `defineCustomElements` to our `main.ts` file.

> Modify `src/main.ts` to reflect the following:

```
import { enableProdMode } from "@angular/core";
import { platformBrowserDynamic } from "@angular/platform-browser-
dynamic";
```

```
import { AppModule } from "./app/app.module";
import { environment } from "./environments/environment";

import { defineCustomElements } from "@ionic/pwa-elements/loader";

if (environment.production) {
  enableProdMode();
}

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

defineCustomElements(window);
```

Set up Ionic Storage

We will be making use of the Ionic Storage API in this application. Ionic provides a simple key/value storage API that we can use in our applications to store data - it provides a consistent API and will automatically use the best storage mechanism available. That means that if we install the SQLite plugin in our project, it will store our data in native storage (rather than browser local storage which is more likely to get wiped by the operating system randomly). If native storage is not available, it will default back to browser storage.

To enable Ionic storage, you need to install it.

> Install Ionic Storage with the following command:

```
npm install @ionic/storage
```

and you will also need to add it to your root module file.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
```

```
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported `IonicStorageModule` and we have also added it to the `imports` array. We have also removed the `SplashScreen` and `StatusBar` providers from Ionic Native since we are not using them anymore.

Set up Additional Modules

As well as the Ionic Storage module, we will also need to make some changes to our home page module. Since we will be launching the Slideshow page as a modal inside of our home page, we will also need to import the `SlideshowPage` here and add it to the `declarations` array. We will also need to add our pipe to the declarations for the home page module as well.

> Modify `src/app/home/home.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { IonicModule } from "@ionic/angular";
import { FormsModule } from "@angular/forms";
import { HomePage } from "./home.page";
import { SlideshowPage } from "../slideshow/slideshow.page";
import { DaysAgoPipe } from "../pipes/days-ago.pipe";
```

```
import { HomePageRoutingModule } from "./home-routing.module";

@NgModule({
  imports: [CommonModule, FormsModule, IonicModule,
HomePageRoutingModule],
  declarations: [HomePage, SlideshowPage, DaysAgoPipe],
})
export class HomePageModule {}
```

Set up Plugins

We are going to set up any native plugins that we need now. Remember that when installing Cordova plugins in a Capacitor project you should install them using the `npm install` command, **not** with `ionic cordova plugin add`.

> Run the following command to add the SQLite plugin:

```
npm install cordova-sqlite-storage
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

Remember, after making changes to the native plugins in your project you will need to run:

```
ionic cap sync
```

This will copy the new plugins you have installed over to the native projects.

Set up Images

When building this application we are going to be making use of a few images. I've included these in your download pack but you will need to set them up in the application you generate.

> Copy the images folder in the download pack for this application from `src/assets` to your own `src/assets` folder

Summary

That's it! We're all set up and ready to go, now we can start working on the interesting stuff. Make sure to test your application by running `ionic serve` every now and then, it's easier to catch and fix errors along the way than waiting until later. If you ever get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again.

Lesson 3: The Layout

I find creating a basic layout is usually a good first step when building an application - it even serves as a sort of wireframing exercise, to help solidify the requirements of the application. There's nothing too complex that we need to do for the layout of this application, but there is one little tricky (and I think clever) user interface element that we are going to add.

We have two pages in this application - the home page and the slideshow page - and we will create both of them in this lesson. The slideshow page is very simple, though, so most of this will be about the home page.

The Home Page

The home page in this application will contain a list of all of the photos that the user has taken with the ability to delete them, an option to take a new photo (but only if they have not already taken one that day), and an option to play a slideshow of all of their photos. Here's what it will look like:

1 days ago

The trickiest part of this layout is that "It's time to take your selfie!" image at the top, which is actually a button that will serve as our take a photo option, but again, only if the user has not already taken a photo. If they have already taken a photo then we don't want to display that image.

Let's start off by creating the basic layout, and then we will look at how we can implement that take photo button.

> Modify `src/app/home/home.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>

    <ion-buttons slot="end">
      <ion-button (click)="playSlideshow()">
        <ion-icon slot="icon-only" name="play"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-list lines="none">
```

```
    <ion-item-sliding>
      <ion-item>
        <img src="https://placehold.it/100x100" />
        <ion-badge slot="end" color="light">0 days ago</ion-badge>
      </ion-item>

      <ion-item-options>
        <ion-item-option color="danger"
(click)="deletePhoto(photo)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

Let's break this template down bit by bit now. The first section of this template is our header:

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>

    <ion-buttons slot="end">
```

```
      <ion-button (click)="playSlideshow()">
        <ion-icon slot="icon-only" name="play"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

The `<ion-toolbar>` allows us to add a bar to the top of our application that can hold buttons, titles, and other elements. We add the `danger` color attribute to the toolbar to style it with our danger colour.

Inside of this toolbar we use `<ion-title>`, which is typically used to display a text title for the current page, to display our logo. We also use `<ion-buttons>` to create a button in the toolbar. By using the `end` slot, the button will be placed on the right side, but if we were to specify the `start` slot it would be displayed on the left instead.

Finally, we have the button itself inside of `<ion-buttons>`. This button uses a play icon and has a click handler attached to it, which will call the `playSlideshow()` function in our **home.page.ts** file (which we have not created yet).

The next part of our template is the content section, which includes a list:

```
<ion-content>
  <ion-list lines="none">
    <ion-item-sliding>
```

```
      <ion-item>
        <img src="https://placehold.it/100x100" />
        <ion-badge slot="end" color="light">0 days ago</ion-badge>
      </ion-item>

      <ion-item-options>
        <ion-item-option color="danger"
(click)="deletePhoto(photo)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

Before we get to the list, notice that everything is wrapped inside of `<ion-content>` - this is what holds the main content for the page and in most cases, everything except the toolbar will be inside of here.

Much like a list is created with plain HTML, e.g:

```
<ul>
  <li></li>
  <li></li>
  <li></li>
</ul>
```

A list in Ionic is created in basically the same way:

```
<ion-list>
  <ion-item></ion-item>
  <ion-item></ion-item>
  <ion-item></ion-item>
</ion-list>
```

Of course, ours looks a little more complicated than that so let's talk through it. The first thing out of the ordinary we are doing is adding `lines="none"` to the `<ion-list>`. Just like the `danger` attribute, which we added to the toolbar, this attribute also styles our list except that it will cause the items in the list not to display with borders.

The next bit gets a little trickier, as it is where we set up our sliding item. An `<ion-item-sliding>`, opposed to a normal `<ion-item>`, has two sets of content - the item itself, and then the `<ion-item-options>` which will be revealed when the user slides the item to the left.

The first block of code inside of `<ion-item-sliding>` is the normal `<ion-item>` definition. Inside of this item will be one of the photos the user has taken (eventually we will create an `*ngFor` loop to display every photo in this list), but for now, we are just using a placeholder image. We also add a badge that will be aligned to the right of the item to display how many days ago the photo was taken. We're also just using dummy data for the "days ago" label, but we will change that later.

Finally, we have the second block of code, `<ion-item-options>`, which allows us to define what content we want to display when the user slides the item. In this case, we are just adding a **Delete** button which will also pass in a reference to the photo it was called on. The `photo` reference it is trying to pass now won't actually work (nor will the function because we haven't created it yet), later on when we create the `*ngFor` loop for all photos we will discuss how to create this reference.

Now we're going to add in that conditional photo button I mentioned earlier.

> Modify the `<ion-content>` section of `src/app/home/home.page.html` to reflect the following:

```
<ion-content>
  <ion-list lines="none">
    <ion-item
      *ngIf="!photoService.photoAlreadyTakenToday"
      button
      detail="false"
      (click)="takePhoto()"
    >
      <img src="assets/images/smile.png" />
    </ion-item>

    <ion-item-sliding>
      <ion-item>
        <img src="http://placehold.it/100x100" />
```

```
        <ion-badge slot="end" color="light">0 days ago</ion-badge>
      </ion-item>

      <ion-item-options>
        <ion-item-option color="danger"
(click)="deletePhoto(photo)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

Notice now that we have included another item in our list above the sliding item, but rather than just using a plain `<ion-item>`, we are also adding the `button` attribute. Visually, this changes nothing, but on mobile everything that is not a `<button>` or `<a>` element that has a click handler will have a slight tap delay. We don't want to have this delay so we add the `button` attribute, indicating that this item is intended to be clicked.

We also have a click handler set up to trigger the `takePhoto` method that we will create later, but the most important thing here is the `*ngIf` structural directive.

`*ngIf` is one of the conditional directives provided by Angular, which allows you to change your template based on some data. In the case of `*ngIf` it will display the element it is attached to, and everything that element contains, only if the expression evaluates to be true. So since we have:

`*ngIf="!photoService.photoAlreadyTakenToday"`

the extra button we added will only display if the `photoService.photoAlreadyTakenToday` value is `false`, since we are essentially saying "display this button if the `photoService.photoAlreadyTakenToday` variable is not true". Later on, we will create a **photoAlreadyTakenToday** variable which will keep track of whether a user has already taken a photo on the current day or not.

The Slideshow Page

As I mentioned, the slideshow page is going to be super simple. It is just going to be a modal that pops up to display all of the user's photos in a quick slideshow. To do this, all we need is a container for the photos, a button to restart the slideshow and a button to close the page.

> Modify `src/app/slideshow/slideshow.page.html` to reflect the following

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>Play</ion-title>

    <ion-buttons slot="start">
      <ion-button (click)="reset()">
        <ion-icon slot="icon-only" name="refresh"></ion-icon>
      </ion-button>
    </ion-buttons>
```

```
    <ion-buttons slot="end">
      <ion-button (click)="close()">
        <ion-icon slot="icon-only" name="close"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <div class="image-container">
    <img [src]="imageSrc" />
  </div>
</ion-content>
```

You already know how the toolbar and buttons work, so the only new thing here is the image player. All we're doing is adding an image element inside of the content area. Later on, we will add some code that will cycle through all the photos the user has taken and change the `src` property to briefly display each photo. To do this, we will just dynamically change the `imageSrc` class member.

But for now, that's all we have to do!

NOTE: Since we are referencing variables in our template that do not exist yet, there will be errors if you try to run the application with `ionic serve` now. You can do this, just keep in mind that the layout issues you see are due to `photoAlreadyTakenToday` not being defined. To see what the application does look like at this stage, you can temporarily remove `*ngIf="!photoService.photoAlreadyTakenToday"` - just make sure to add

it back afterward.

Summary

We now have the templates set up for both of the pages in our application, so in the next lesson, we can start diving into some more interesting stuff. In fact, it will probably be one of the most interesting lessons because we will be learning how to integrate with the `Camera` API to take photos and using the `Filesystem` API to move them around!

Lesson 4: Taking Photos with the Camera

The primary goal of this lesson is to integrate the Camera to allow users to take photos, but we are going to be doing quite a bit to make that work.

Of course, we need to trigger the camera to take a photo, but we also need to:

- Move the photo to permanent storage on the phone
- Display the photo in our template
- Display all the other photos
- Allow the photo to be deleted

This is going to be a pretty large lesson. During this lesson, we will also be making use of the `Filesystem` API and `Share` API from Capacitor. Let's get started!

Creating a Simple Alert Service

Before we get into the fun Camera stuff, I've got a little task that we are going to complete first. We're going to be triggering different alerts in this application because there are a few different places where errors can occur that the users need to be notified about, and we will also be alerting the user when things have gone well as well.

To create an alert, the syntax looks this:

```
const alert = await this.alertCtrl.create({
```

```
  header: 'My header',
  message: 'My message',
  buttons: [
    {
      text: 'Ok'
    }
  ]
});

alert.present();
```

Which isn't *too* much code, but if we are frequently creating alerts with this same basic structure it can get a bit tedious to constantly create them with the `AlertController` manually. We're going to create a service that will handle creating and displaying the alert for us. Once we have created it, we will instead be able to create and display an alert automatically by doing this:

```
this.simpleAlertService.showAlert('Oops!', 'Some message');
```

Doing something like this isn't really required, but it does save us a little effort in the long run as we don't need to keep recreating the alert structure. However, this is also restricting because all we are able to do now is change the `header` and `message` values. If you we wanted a different style of alert, one that had *two* buttons for example, then we would still need to use the `AlertController` directly.

> **Modify `src/app/services/simple-alert.service.ts` to reflect the following:**

```
import { Injectable } from "@angular/core";
import { AlertController } from "@ionic/angular";

@Injectable({
  providedIn: "root",
})
export class SimpleAlertService {
  constructor(private alertCtrl: AlertController) {}

  async showAlert(title: string, message: string): Promise<void> {
    const alert = await this.alertCtrl.create({
      header: title,
      message: message,
      buttons: [
        {
          text: "Ok",
        },
      ],
    });

    await alert.present();
  }
}
```

We've imported the `AlertController` service here so that we can create an alert, and then we have just created a single method that takes in a `title` and a `message` and then creates and displays an alert using those values.

To use this simple alert service, we can just import and inject it in whatever page we want to use it in.

> Modify `src/app/home/home.page.ts` to reflect the following:

```
import { Component, OnInit } from "@angular/core";
import { SimpleAlertService } from "../services/simple-
alert.service";

@Component({
  selector: "app-home",
  templateUrl: "./home.page.html",
  styleUrls: ["./home.page.scss"]
})
export class HomePage implements OnInit {
  constructor(private simpleAlertService: SimpleAlertService) {}

  ngOnInit() {}
}
```

After injecting the service into the home page as we have above, we would be able to access it anywhere in the class through `this.simpleAlertService`.

Setting up the Photo Interface

Before we can start creating photos, we will need to implement our `Photo` interface that defines the structure of a photo object.

> Modify `src/app/interfaces/photo.ts` to reflect the following:

```
export interface Photo {
  name: string;
  path: string;
  dateTaken: Date;
}
```

All we will be storing for our photo is the file `name` we create for it, the `path` to where it is stored in the native file system (or for our web version, a `base64` string of the photo instead), and the `dateTaken` which will indicate the time at which the photo was created.

Remember, custom types like this are not required, but they can help prevent issues in your application by ensuring that your data conforms to the expected structure and also help provide autocompletion hints within editors like Visual Studio Code.

Taking a Photo with the Camera

Ok, now we can finally focus on the fun stuff which is actually taking the photo. To avoid cluttering up our home page with functionality related to accessing and storing photos, we

are going to implement the bulk of the logic in a separate photo service. In general, it is a good idea to keep your pages light and move any kind of complicated work out into services.

We are going to start setting up our home page, and then we will switch over to building our photo service when required. Before we add the code for taking a photo, we are going to set up some imports and inject some dependencies through the constructor - we are going to just set up everything we will need now so we don't have to keep coming back and modifying our imports all the time.

> Modify `src/app/home/home.page.ts` to reflect the following:

```
import { Component, OnInit, ViewChild } from "@angular/core";
import { DomSanitizer } from "@angular/platform-browser";
import {
  IonList,
  LoadingController,
  AlertController,
  ModalController,
  Platform,
} from "@ionic/angular";
import { Plugins } from "@capacitor/core";
import { SimpleAlertService } from "../services/simple-
alert.service";
import { PhotoService } from "../services/photo.service";
import { Photo } from "../interfaces/photo";
import { SlideshowPage } from "../slideshow/slideshow.page";
```

```
const { Share } = Plugins;

@Component({
  selector: "app-home",
  templateUrl: "home.page.html",
  styleUrls: ["home.page.scss"],
})
export class HomePage implements OnInit {
  @ViewChild(IonList, { static: false }) slidingList: IonList;

  public photos: Photo[] = [];

  constructor(
    public photoService: PhotoService,
    public sanitizer: DomSanitizer,
    private loadingCtrl: LoadingController,
    private alertCtrl: AlertController,
    private modalCtrl: ModalController,
    private simpleAlertService: SimpleAlertService,
    private platform: Platform
  ) {}

  ngOnInit() {
    this.photoService.getPhotos().subscribe((photos) => {
      this.photos = photos;
    });
  }
```

```
  async takePhoto(): Promise<void> {}

  async playSlideshow(): Promise<void> {}

  async deletePhoto(photo: Photo): Promise<void> {}
}
```

We've set up the basic skeleton for our home page here. We have a bunch of imports at the top that we will be making use of later. We have also set up a reference to the `Share` API from Capacitor, which we will use to give the user the option to share any photos that they take.

We use our `ngOnInit` (which runs as soon as this component is initialised) to subscribe to an observable that our photos service will provide for us. This will allow us to update `this.photos` with new photos as soon as they appear in the photos service. We will trigger the actual taking of the photo through the `takePhoto` method, and we also have a `playSlideshow` method which will launch our slideshow page to display the slideshow of photos to the user.

We have also included `@ViewChild` and `IonList` just like we did in the QuickLists application so that we will be able to grab a reference to the list and close any sliding items before we delete a photo.

You might also notice that we are still importing `AlertController` as well as our own `SimpleAlertService`. We will also be creating a more complex alert later, so we will also be creating an alert directly through the `AlertController`, and for the simple alerts, we will use our `SimpleAlertService`.

Something a little weird you might take notice of here is that the inclusion of `DomSanitizer`. We will use this throughout the application to create a `SafeResourceUrl`, which represents a resource that we are telling Angular is safe to access. The reason we need to do this is that we want to display a photo that is going to be stored locally on the device, and we will be accessing that through a resource URL. When encountering a resource URL like this, Angular will block it by default for security reasons. What we need to do is manually bypass this. You will see in just a moment that what we do is we get the resulting File URI from the Camera API, we then use the `convertFileSrc` method that Capacitor provides to convert that URI into a format that Capacitor can access safely, and then we will let Angular know that this is a "safe resource URL" by calling the `bypassSecurityTrustResourceUrl` on that path when we try to display it. If we don't do this, Angular will prevent any attempt at trying to access the photo.

Before we can move much further, we need to start implementing our photo service. Let's start setting that up now.

> Modify `src/app/services/photo.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { Platform } from "@ionic/angular";
import { Storage } from "@ionic/storage";

import {
  Plugins,
  Capacitor,
```

```
  CameraResultType,
  CameraSource,
  FilesystemDirectory,
  CameraOptions,
  CameraPhoto,
  FileReadResult,
  FileWriteResult,
} from "@capacitor/core";

import { BehaviorSubject, Observable } from "rxjs";
import { Photo } from "../interfaces/photo";

const { Camera, Filesystem } = Plugins;

@Injectable({
  providedIn: "root",
})
export class PhotoService {
  public photoAlreadyTakenToday: boolean = false;

  private photos$: BehaviorSubject<Photo[]> = new
BehaviorSubject<Photo[]>([]);
  private photos: Photo[] = [];
  private loaded: boolean = false;

  constructor(private storage: Storage, private platform:
Platform) {}

  async load(): Promise<void> {
```

```
      // Uncomment to use test data instead of real data
      /*this.photos = [
        { name: "test", path: "https://placehold.it/100x100",
dateTaken: new Date(2018, 5, 5) },
        { name: "test", path: "https://placehold.it/110x110",
dateTaken: new Date(2018, 5, 6) },
        { name: "test", path: "https://placehold.it/120x120",
dateTaken: new Date(2018, 5, 8) },
        { name: "test", path: "https://placehold.it/130x130",
dateTaken: new Date(2018, 5, 10) },
      ];*/

      this.checkIfPhotoAlreadyTakenToday();
      this.photos$.next(this.photos);

      this.addPhotoTakenListener();
  }

  getPhotos(): Observable<Photo[]> {
    return this.photos$;
  }

  checkIfPhotoAlreadyTakenToday(): void {
    if (this.photos.length > 0) {
      const today = new Date();
      const latestPhotoDate = new Date(this.photos[0].dateTaken);

      if (latestPhotoDate.setHours(0, 0, 0, 0) ===
today.setHours(0, 0, 0, 0)) {
```

```
        this.photoAlreadyTakenToday = true;
      } else {
        this.photoAlreadyTakenToday = false;
      }
    } else {
      this.photoAlreadyTakenToday = false;
    }
  }

  addPhotoTakenListener(): void {
    // We need to listen for the "resume" event in case the user
has the app running for
    // more than a full day in the background
    this.platform.resume.subscribe(() => {
      this.checkIfPhotoAlreadyTakenToday();
    });
  }

  async takePhoto(): Promise<string> {
    return "";
  }

  createPhoto(fileName: string, filePath: string): void {}

  async deletePhoto(photoToDelete: Photo): Promise<void> {}

  save(): Promise<void> {
    return Promise.resolve()
  }
```

```
}
```

We've set up a basic skeleton for our photos service for now, but there is also a little bit more going on here already. We've got some pretty standard imports, and we have also set up a reference to the two plugins we will be using in this service:

```
const { Camera, Filesystem } = Plugins;
```

We can use these to access `Camera` and `Filesystem` throughout the service. Let's also address why there are so many imports related to the `Camera` and `Filesystem`:

```
import {
  Plugins,
  Capacitor,
  CameraResultType,
  CameraSource,
  FilesystemDirectory,
  CameraOptions,
  CameraPhoto,
  FileReadResult,
  FileWriteResult,
} from "@capacitor/core";
```

The only imports we actually *need* here are `Plugins` and `Capacitor`. The only reason we are included the rest of them is so that we can assign types to results we are getting from the Camera and Filesystem plugins. The `Filesystem` especially is a bit complicated, and by assigning a type like `FileReadResult` to a variable that we are using to hold a result from a file read, we can much more easily see what sorts of methods are available on it. Of course, as I keep mentioning, we also have the benefit of type safety which ensures that the data we are assigning to the variable is actually a `FileReadResult` as we are expecting.

We've set up a few member variables in this service. Just like in the previous two applications, we have a variable called `photos` to hold our array of data that we will be manipulating in this service, and we have a variable called `photos$` which is a `BehaviorSubject`/`Observable`. The `photos` variable will be used internally to the service to manage the photo data, and the `photos$` observable will expose that data throughout the rest of the application such that we can subscribe to it for updates to photos. You've already seen that we are creating this subscription in the home page to update `this.photos`. We also have a `photoAlreadyTakenToday` boolean which keeps track of whether or not the user has already taken their photo for the day, and `loaded` boolean to keep track of whether or not the photo data has been loaded in from storage (we will handle this in the next lesson).

We've set up some test data inside of the `load` method so that we can test our layout without having to actually take photos first. If you need to do any testing with photos present, just uncomment that section (make sure to comment it out again later!).

We've also added a weird `resume` listener here inside of `addPhotoTakenListener`. The `resume` event will fire whenever a person sends your application to the background and

then resumes using it again later. For example, they have your application open, switch to Facebook, and then back to yours. The reason we're doing this is because there's a bit of an edge case with our `photoAlreadyTakenToday` variable. Imagine someone has taken their photo for the day, but when they close the application they don't fully close it, it just goes to the background. When they come back the next day the `load` method won't run again because the application has already been started. Therefore, if we are only running the logic once inside of `load` to check if the photo was taken or not, it won't work in this case. So whenever the application is resumed, we check if the date of the last photo taken is equal to today's date, and then set the `photoAlreadyTakenToday` variable according to that. We trigger this logic once manually when the application first loads with `this.checkIfPhotoAlreadyTakenToday()` and then we call `this.addPhotoTakenListener()` to start listening for `resume` events.

Now we're going to implement the `takePhoto` method which will handle taking the photo. Let's create a basic version of that first.

> Modify the `takePhoto` method in `photo.service.ts` to reflect the following:

```
async takePhoto(): Promise<string> {
  if (!this.loaded || this.photoAlreadyTakenToday) {
    throw new Error("Not allowed to take photo");
  }

  const options: CameraOptions = {
    quality: 50,
    width: 600,
```

```
      allowEditing: false,
      resultType: this.platform.is("capacitor") ?
CameraResultType.Uri : CameraResultType.DataUrl,
      source: CameraSource.Camera,
    };

    try {
      const photo: CameraPhoto = await Camera.getPhoto(options);
      console.log(photo);
    } catch (err) {
      throw new Error("Could not write file");
    }
  }
```

Before we execute the `Camera` code, we check a few conditions first. If the variables we just recently created indicate that the data has not finished loading yet, or that a photo has already been taken today, then we `throw` an error to stop executing the rest of the method.

Then we set up some options to pass to the `Camera` plugin, these configure what exactly we want to do and what we want returned. These values can configure things like whether to use the camera or the user's photo library, the format to return the image in, whether to use the back-facing or front-facing camera and so on. Notice that we give these options a type of `CameraOptions`, this will make sure that we construct our `options` object correctly and if we are using something like Visual Studio Code it will also give us autocompletion hints.

In this case, we want to use the camera to retrieve a photo and we want the photo to be returned as a URI, which means we will be given a path to the photo on the device (rather than the photo data itself). However, if `this.platform.is("capacitor")` is not `true` (i.e. the application is running on a native device), then we want the result returned as a `DataUrl` instead for the web. Instead of a path to the file on the device, this will give us a `base64` data string that can be embedded directly into an image (the entire photo itself is encoded into this `base64` string). This means that we won't need to deal with the filesystem like we would with a URI, but storing photos as `base64` strings in browser storage is not a good long term solution for photo storage, which is why we would only use the web version for testing (unless we modified it to use some kind of cloud based storage instead for the web).

Once we've created that options object, we call the `getPhoto` method on the `Camera` object and pass in the options. This will return a promise which, when resolved, will give us a path to the image on the user's device. Right now we are just logging out that value but we're about to do a lot more with it. If an error value is returned then we will throw our own error indicating that the file could not be written (we aren't actually *trying* to write the file just yet, but we will in a moment).

The image that is taken with the camera, and the path that is returned, is in a **temporary** storage folder. The image will display for a little while but if we want to keep the photo around more permanently it's not going to work, because eventually it will just be deleted. To solve this issue, we are going to have to take that photo and move it somewhere else, and then we are going to store a reference to *that* location for the photo (although, I would still advise against relying just on local file storage if it is critical that the photo is never lost, even if you are using native storage). We will do this using the `Filesystem` plugin.

Moving the Photo to Permanent Storage

In order to use the Filesystem API plugin, we're going to modify the `takePhoto` method to do the following:

- Use the `path` returned to find the photo in temporary storage
- Rename and move it to a permanent folder
- Create a new photo object in the application based on the new location of the photo

Let's modify the function and then talk through it.

> Modify the `takePhoto` method in `photo.service.ts` to reflect the following:

```
  async takePhoto(): Promise<string> {
    if (!this.loaded || this.photoAlreadyTakenToday) {
      throw new Error("Not allowed to take photo");
    }

    const options: CameraOptions = {
      quality: 50,
      width: 600,
      allowEditing: false,
      resultType: this.platform.is("capacitor") ?
CameraResultType.Uri : CameraResultType.DataUrl,
      source: CameraSource.Camera,
    };
```

```
    try {
      const photo: CameraPhoto = await Camera.getPhoto(options);

      if (this.platform.is("capacitor")) {
        const photoOnFileSystem: FileReadResult = await
Filesystem.readFile({ path: photo.path });

        const fileName = Date.now().toString() + ".jpeg";
        const permanentFile: FileWriteResult = await
Filesystem.writeFile({
          data: photoOnFileSystem.data,
          path: fileName,
          directory: FilesystemDirectory.Data,
        });

        this.createPhoto(fileName,
Capacitor.convertFileSrc(permanentFile.uri));
        this.photoAlreadyTakenToday = true;
        return photo.path;
      } else {
        this.createPhoto(Date.now().toString(), photo.dataUrl);
        this.photoAlreadyTakenToday = true;
        return photo.dataUrl;
      }
    } catch (err) {
      throw new Error("Could not write file");
    }
  }
```

That's quite a bit more complicated now. Before we discuss what is happening with the file system, notice that we perform different logic depending on the result of `if (this.platform.is("capacitor"))`. If we are running natively on iOS/Android then we want to interact with the file system, but if we are on the web, then we want to just instantly create a new photo using the `base64` data URL that was returned.

The code above that is dealing with the file system is working through a progression of reading and writing files. Let's dot point what exactly is happening step-by-step.

1. Grab a reference to the file on the device by passing the `readFile` method the path that the Camera API gave to us
2. Create a new file name based on the date, and then write that file using the data we just read from the photo to the `Data` directory.
3. Call the `createPhoto` method to create a new `Photo` object by passing in the new `fileNName` and the URI to the file in the permanent location, first making sure to convert it to a Capacitor friendly URL (if we don't call `convertFileSrc` then Capacitor will not be able to access the photo via the URI)

After all of this we should have the photo stored in a permanent location, along with a path to the image in its new location. Keep in mind that we haven't actually created the `createPhoto` function yet. We're going to do that now.

The `createPhoto` function will take the path (the URI) and keep a reference to it in the application so we can find it later.

> Modify the `createPhoto` method to reflect the following:

```
createPhoto(fileName: string, filePath: string): void {
  const newPhotos = [
    {
      name: fileName,
      path: filePath,
      dateTaken: new Date(),
    },
    ...this.photos,
  ];

  this.photos = newPhotos;

  this.save();
}
```

All we are doing here is creating a new array containing all of our old photos (using the same immutability concepts that we used in the QuickLists lessons), but we also add a new photo object at the beginning of the array.

We also have a call to the `save` function, which will save our data into local storage but we haven't implemented that functionality yet. Note that what we will be storing in local storage is simply a path to where the photo is physically stored on the device, not the photo itself. This way, we will only be storing a small amount of text data in local storage, and the big image files are on the file system.

Although we will be implementing the data storage aspect later, if we want the rest of our

application to be notified about this new photo we will need to trigger the `next` method on our `photos$` `BehaviorSubject`/`Observable`.

> Modify the save method to reflect the following:

```
  save(): Promise<void> {
    this.photos$.next(this.photos);
    return Promise.resolve();
  }
```

If we have a way to create photos, then we will also want a way to delete them. Not only do we want to delete them from the application, but we want to make sure the photo is removed from storage as well.

> Modify the `deletePhoto` method to reflect the following:

```
  async deletePhoto(photoToDelete: Photo): Promise<void> {
    // Remove data from storage
    this.photos = this.photos.filter((photo) => photo.name !==
photoToDelete.name);
    this.save();

    // If the deleted photo was taken today, allow the user to
take a new photo
    const today = new Date();
```

```
    const dateTaken = new Date(photoToDelete.dateTaken);

    if (dateTaken.setHours(0, 0, 0, 0) === today.setHours(0, 0, 0,
0)) {
      this.photoAlreadyTakenToday = false;
    }

    if (this.platform.is("capacitor")) {
      await Filesystem.deleteFile({
        path: photoToDelete.name,
        directory: FilesystemDirectory.Data,
      });
    }
  }
```

First, we remove the photo from the `photos` array and save that data to remove the reference to the photo. We also want to allow the user to take another photo if the photo they deleted was taken today, so we add a check for that as well. Finally, we use the `Filesystem` plugin again to remove the photo from the device. We do this by passing in the `path` to the photo (which will just be its name within the `Data` directory), and the `directory` that it was stored in. Note that we only run this code if we are running natively on a device, if we are using the web version then simply removing the entry from the `photos` array will be sufficient.

Updating the Template

We have photos being added to our `this.photos` array now, so now we can update our

template to loop through and display all of them. Feel free to uncomment the test data we added before so that you can see the results of the code in this section.

> Modify `src/app/home/home.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>

    <ion-buttons slot="end">
      <ion-button (click)="playSlideshow()">
        <ion-icon slot="icon-only" name="play"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-list lines="none">
    <ion-item
      *ngIf="!photoService.photoAlreadyTakenToday"
      button
      detail="false"
      (click)="takePhoto()"
    >
```

```
      <img src="assets/images/smile.png" />
    </ion-item>

    <ion-item-sliding *ngFor="let photo of photos">
      <ion-item>
        <img
[src]="sanitizer.bypassSecurityTrustResourceUrl(photo.path)" />
        <ion-badge slot="end" color="light">0 days ago</ion-badge>
      </ion-item>

      <ion-item-options>
        <ion-item-option color="danger"
(click)="deletePhoto(photo)">
          <ion-icon slot="icon-only" name="trash"></ion-icon>
        </ion-item-option>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

There are two main additions to the template here. First, we are using `*ngFor` to loop through our array of photos and create an `<ion-item-sliding>` entry for all of them. Using `let photo` in `let photo of photos` allows us to reference the specific photo that is being rendered in the loop by using `photo`, e.g: `photo.path` to access the image path stored on the photo.

Since we can access the path of each photo, we can now also set the `<img>` element to

display it. Notice that we use square brackets around `[src]`, this allows us to set the **src** property to the expression `photo.path` on the image element. This means that `photo.path` will first be evaluated to whatever value it stores (e.g. `/path/to/image`), and then set as the **src**. If we didn't use the square brackets, then the **src** would literally be set to the string "photo.path". We also surround `photo.path` with a call to the `bypassSecurityTrustResourceUrl` method from the `DomSanitizer` which will convert the path into a `SafeResourceUrl` for us, which will allow us to access it.

We also have a delete button here that passes a reference to the photo (which we created by using `let photo`) through to the `deletePhoto` function. We will need to make sure to define that in our **home.page.ts** file.

> Modify the `deletePhoto` method in `src/app/home/home.page.ts` to reflect the following:

```
async deletePhoto(photo: Photo): Promise<void> {
  await this.slidingList.closeSlidingItems();
  this.photoService.deletePhoto(photo);
}
```

The last thing we need to do is implement the `takePhoto` method in our home page so that it actually makes a call to the `takePhoto` method we just implemented in the photo service.

> Modify the `takePhoto` method in `src/app/home/home.page.ts` to reflect the

following:

```
  async takePhoto(): Promise<void> {
    const loadingOverlay = await this.loadingCtrl.create({
message: "Saving photo..." });
    loadingOverlay.present();

    try {
      const photoUri = await this.photoService.takePhoto();
      loadingOverlay.dismiss();

      if (this.platform.is("capacitor")) {
        const alert = await this.alertCtrl.create({
          header: "Nice one!",
          message: "You've taken your snap for today, would you
also like to share it?",
          buttons: [
            {
              text: "No",
            },
            {
              text: "Share",
              handler: () => {
                Share.share({
                  title: "Today's snap",
                  text: "I'm taking a selfie every day with
#Snapaday",
```

```
                  url: photoUri,
                });
              },
            },
          ],
        });

        alert.present();
      } else {
        this.simpleAlertService.showAlert("Nice one!", "You've
taken your snap for today.");
      }
    } catch (err) {
      loadingOverlay.dismiss();
      this.simpleAlertService.showAlert(
        "Oops!",
        "Your photo could not be added, please try again."
      );
      console.warn(err);
    }
  }
```

We are doing a little more than just calling `this.photoService.takePhoto()` here. Since the process of taking the photo and then saving into storage will take a few seconds, we want to pop up a loading overlay to indicate to the user that something is happening - which we do by using the `loadingCtrl` to create the overlay, and then when it is ready we call `loadingOverlay.present()`.

At this point, we make a call to the `takePhoto` method in our photo service and we set up a handler for the photo we get in return. At this point, the photo has finished saving so we dismiss the overlay. However, we also want to give the user the option to share this photo. We pop up a custom alert to give them that option, and if they choose the `Share` option we trigger the `Share` plugin that we are using from Capacitor. We supply the photo data to the plugin, and from there the plugin handles the rest - it will pop up an overlay for the user to choose their preferred method of sharing. Note that if we are on the web platform, we don't offer the sharing functionality as `Share` does not (yet) have a web implementation.

Before we conclude this lesson, we are going to add a call to our photo services `load` method in our root component so that the load begins as soon as the application opens.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { PhotoService } from "./services/photo.service";

const { SplashScreen, StatusBar, LocalNotifications } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(private photoService: PhotoService) {
```

```
    this.initializeApp();
  }

  async initializeApp() {
    await this.photoService.load();

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#eb445a" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

Summary

What a lesson! There was some pretty complex stuff covered in this lesson but we've implemented the core functionality of the application now. There's still quite a bit left to do, but we can take photos with the camera and see them displayed in a list which is pretty cool. To see all of the features for yourself you will need to run it on a real device (however, you will still be able to take photos through the web browser just by running `ionic serve`). If you don't know how to get the application running on a native device already, you can skip to the **Building & Submitting** section of this book to see how to get the app installed on your device, and then come back to finish the rest of the application.

IMPORTANT: In order for our application to allow a photo to be taken it checks if the data has been loaded. We haven't done this yet, so if you run the application you won't be able to take a photo. To circumvent this temporarily, you can just add `this.loaded = true` to the top of the `load()` method in the photo service.

Lesson 5: Saving and Loading Photos

We can now take photos in the application and display those photos in a list, we can even delete them as well. The problem is that the data is stored on the `this.photos` array and will be lost as soon as the application is closed. Obviously, the user isn't going to be very happy when they find their selfie missing the next time they try to use the application, so in this lesson, we are going to store this data permanently.

We've already got a lot of this process set up, we've already got the `Storage` API set up in our photo service, and we even have a `save` function created that we are calling whenever we want data saved. All we really have to do is:

- Implement the save function so that it uses the `Storage` API
- Modify our `load` method in the photo service to pull in existing photos from storage

If you don't recall from the earlier chapters, or you haven't read them yet, **Storage** is Ionic's generic storage service. It handles storing data in the best way possible whilst providing a consistent API for us to use.

When running on a device, and if the SQLite plugin is available (which we installed earlier), it will store data using a native SQLite database. Since the SQLite database will only be available when running natively on a device, `Storage` will also use `IndexedDB`, `WebSQL`, or standard browser `localStorage` if the SQLite database is not available.

It's best to use SQLite where possible because the browser-based local storage is not

completely reliable and can potentially be wiped by the operating system. Having your data wiped randomly is obviously not ideal.

Implementing Data Storage

To handle loading any stored data into the application we're going to modify the `load` method in our photo service. We are already calling this method from our root component when the application launches, so whatever is in `load` will get triggered right away.

> Modify the `load` method in `src/app/services/photo.service.ts` to reflect the following:

```
  async load(): Promise<void> {
    if (!this.loaded) {
      const photos = await this.storage.get("photos");
      if (photos !== null) {
        this.photos = photos;
      }

      // Uncomment to use test data instead of real data
      /*this.photos = [
        { name: "test", path: "https://placehold.it/100x100",
dateTaken: new Date(2018, 5, 5) },
        { name: "test", path: "https://placehold.it/110x110",
dateTaken: new Date(2018, 5, 6) },
        { name: "test", path: "https://placehold.it/120x120",
dateTaken: new Date(2018, 5, 8) },
```

```
        { name: "test", path: "https://placehold.it/130x130",
dateTaken: new Date(2018, 5, 10) },
      ];*/

      this.checkIfPhotoAlreadyTakenToday();
      this.photos$.next(this.photos);

      this.addPhotoTakenListener();

      this.loaded = true;
    }
  }
```

All we are doing differently now is first checking to see if `loaded` is not set to `true` yet (this is just an additional check to make sure we don't call the `load` method more than once from somewhere in our application), and then we grab whatever is stored on the `photos` key in storage. If this value is not `null` (i.e. there is some data available) we use that data for our `this.photos` array. If there was no data in storage, then the `this.photos` array would just remain as an empty array. At the bottom of the method, we then set the `loaded` flag to true.

Now all we have left to do is make our `save` method actually store the photo data to storage.

> Modify the `save` method in `src/app/services/photo.service.ts` to reflect the following:

```
save(): Promise<void> {
  this.photos$.next(this.photos);
  return this.storage.set("photos", this.photos);
}
```

Now whenever the `save` method is called it will pass in all the current values in `this.photos` and save them into storage.

Summary

That lesson was certainly a lot shorter, so hopefully, you enjoyed a bit of a break after the last one. Saving and loading data sounds like a pretty complex topic, but as you can see storing simple data is actually quite easy.

In the next lesson, we're going to get back to doing something a little more fun, we're going to finish off the Slideshow page so that it shows a slideshow of all of the user's photos, as well as implementing our own custom "Days Ago" pipe.

Lesson 6: Custom Pipe and Flipbook

In this lesson, we will be creating our own custom pipe to display the dates photos were taken in a more user-friendly way, and we will also be finishing off a critical portion of our application which is the slideshow. It won't be quite as complex as actually taking the photo was, but the slideshow is pretty much the point of the whole application.

Creating a Custom Pipe

We've already covered what a pipe is in the basics section of this book, but to refresh your memory a pipe essentially allows us to modify some data before displaying it to the user in the template.

The goal for us here is to create a label that displays how many days ago a photo was taken, e.g. "3 days ago", "10 days ago". Right now, the only data we have stored on our photo that we can use is a date object, unfortunately for us, that looks something like this:

```
Sun Mar 06 2016 00:40:02 GMT+1000 (AEST)
```

which isn't the most user-friendly format in the world. So a `@Pipe` is a perfect use case for this: we create a pipe that takes in our ugly date format, converts it into the "X days ago" format, and then returns it.

Let's take a look at how to create the pipe first, and then we'll go over how to use it.

> Modify `src/app/pipes/days-ago.pipe.ts` to reflect the following:

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

@Pipe({
  name: "daysAgo",
})
export class DaysAgoPipe implements PipeTransform {
  transform(value: string, ...args: unknown[]): unknown {
    const now = new Date();
    const takenDate = new Date(value);
    const oneDayInMs = 24 * 60 * 60 * 1000;
    const diffDays = Math.round(Math.abs((takenDate.getTime() -
now.getTime()) / oneDayInMs));
    return diffDays;
  }
}
```

In the **@Pipe** decorator we supply a `name` of `daysAgo`, this means that this pipe can be used by using `daysAgo` as the keyword in the template. When using a pipe, you always pass data into it, and this data gets sent to the `transform` function and is passed in as the `value`. We will be passing the `Date` value from our photo into this pipe, so that's what we will be working with here. Also note that additional arguments *can* also be supplied through the pipe using the `args` parameter.

First, we do a little mathematics magic to work out the difference in days between the current date and the date the photo was taken (Stack Overflow gets the credit for this one, my solution would have been way uglier than this), and then we return it. Whatever value is

returned is what will actually be rendered out to the user in the template.

Since we have already set this pipe up in our application (in the **home.module.ts** file) we can go right ahead and use it now.

> Modify the photo item in `src/app/home/home.page.html` to reflect the following:

```
      <ion-item>
        <img
[src]="sanitizer.bypassSecurityTrustResourceUrl(photo.path)" />
        <ion-badge slot="end" color="light">{{ photo.dateTaken |
daysAgo }} days ago</ion-badge>
      </ion-item>
```

Notice that we have added the following:

`{{photo.dateTaken | daysAgo}}`

This will pass `photo.dateTaken` into the `daysAgo` pipe, and then whatever `daysAgo` returns will be displayed to the user here. The end result will be a badge containing something like "5 days ago".

Creating a Slideshow of All Photos

We've already created the template for the Slideshow, so now we need to create some

logic so that all of the photos are cycled through and displayed, and we will also need to add a way to open the slideshow page (as well as restart the slideshow).

Let's start off by defining the `playSlideshow` method so that we can actually open the page.

> Modify the `playSlideshow` method in `src/app/home/home.page.ts` to reflect the following:

```
  async playSlideshow(): Promise<void> {
    if (this.photos.length > 1) {
      const modal = await this.modalCtrl.create({
        component: SlideshowPage,
      });

      modal.present();
    } else {
      this.simpleAlertService.showAlert(
        "Oops!",
        "You need at least two photos before you can play a
slideshow."
      );
    }
  }
```

We display a modal page to users very similarly to how we create alerts. We create a

modal using the `ModalController`, and we pass it the component/page that we want to display. In this case, we create a Modal using the `SlideshowPage` that we have already created.

Notice that we only trigger the modal if the user has more than `1` photo (because a slideshow with `0` or `1` photos isn't really a slideshow, right?), and if they don't an alert is displayed.

Now we are going to define the class for the **Slideshow** page. The class is reasonably small so I'm going to post all of the code in one block and then we will step through it afterward.

> Modify `src/app/slideshow/slideshow.page.ts` to reflect the following:

```
import { Component, OnDestroy, OnInit } from "@angular/core";
import { SafeResourceUrl, DomSanitizer } from "@angular/platform-
browser";
import { ModalController } from "@ionic/angular";
import { SimpleAlertService } from "../services/simple-
alert.service";
import { Photo } from "../interfaces/photo";
import { PhotoService } from "../services/photo.service";
import { take } from "rxjs/operators";

@Component({
  selector: "app-slideshow",
  templateUrl: "./slideshow.page.html",
```

```
  styleUrls: ["./slideshow.page.scss"],
})
export class SlideshowPage implements OnInit, OnDestroy {
  public imageSrc: SafeResourceUrl;
  private photos: Photo[];
  private slideshowInterval: ReturnType<typeof setInterval>;

  constructor(
    private modalCtrl: ModalController,
    private photoService: PhotoService,
    private sanitizer: DomSanitizer,
    private simpleAlertService: SimpleAlertService
  ) {}

  ngOnInit() {
    this.photoService
      .getPhotos()
      .pipe(take(1))
      .subscribe((photos) => {
        this.photos = photos;
        this.reset();
      });
  }

  ngOnDestroy() {
    clearInterval(this.slideshowInterval);
  }

  playSlideshow(): void {
```

```
    let currentPhotoIndex = 1;

    //Clear any interval already set
    clearInterval(this.slideshowInterval);

    this.slideshowInterval = setInterval(() => {
      if (currentPhotoIndex < this.photos.length) {
        this.imageSrc =
this.sanitizer.bypassSecurityTrustUrl(this.photos[currentPhotoIndex

        currentPhotoIndex++;
      } else {
        clearInterval(this.slideshowInterval);
      }
    }, 1000);
  }

  reset(): void {
    if (this.photos.length > 1) {
      this.imageSrc =
this.sanitizer.bypassSecurityTrustUrl(this.photos[0].path);
      this.playSlideshow();
    } else {
      this.simpleAlertService.showAlert(
        "Oops!",
        "You need at least two photos to play the slideshow"
      );
    }
  }
```

```
  close(): void {
    this.modalCtrl.dismiss();
  }
}
```

The most important method here is the `playSlideshow` method. Basically, we want to loop through each of our photos and continually change `imageSrc` so that it displays each photo in succession. We clear any intervals that are currently running (in case a user restarts the slideshow when it is still currently playing) and then we start looping through all of the photos in `this.photos`. If there are still photos left to display, the image elements **src** property is updated with the next photo, and if there aren't photos left to display then the interval is cleared. In the code above the interval is set to run every `1000ms` or once every second, so you could adjust this to make the slideshow slower or faster if you want (or perhaps even add an option for the user to control that). Again, we need to make use of the `DomSanitizer` and a `SafeResourceUrl` here otherwise Angular will block the attempt to access the photo.

Notice that we have also set up a call to `clearInterval` in `ngOnDestroy` to destroy our interval if the user closes the modal before the slideshow has ended. If we didn't do this, the interval would continue running in the background until the slideshow has completed, even though the user is not even viewing it anymore.

We are also doing something a little trick in our `ngOnInit` lifecycle hook:

```
ngOnInit() {
  this.photoService
    .getPhotos()
    .pipe(take(1))
    .subscribe((photos) => {
      this.photos = photos;
      this.reset();
    });
}
```

Before we trigger our slideshow to start by calling the `reset` method (we use this method both to start the slideshow initially, and restart it every time the user triggers the `reset` method) we need to get access to the photos we want to display. To do this, we subscribe to the observable the photos service provides just like we did on the home page. However, we don't need an active subscription that will constantly update as the photos change, we just need the data once and that's it - we don't want to leave any unnecessary subscriptions lingering around our application. To make this easy, we use the `take` operator. By piping on `take(1)` the observable will only emit a value once (whatever photos are currently available in our photo service) and then it will automatically unsubscribe.

The only other strange thing going on here is this type:

```
private slideshowInterval: ReturnType<typeof setInterval>;
```

We are taking care to give types to everything so we can benefit from type safety and autocompletion hints. We don't *have* to give types to everything, so we could just skip this one since it is difficult, but once you start making exceptions its tempting to start using `any` a little too often. We store a reference to the interval we create with `setInterval` so that we can clear the interval when it is no longer needed (otherwise it will just keep running forever in the background, and this could lead to problems). The type of this reference is a bit hard to specify though because there is a NodeJS interval available, and also one defined on the `window` object of the browser. We are using the browsers interval in this case, but the code above is a more platform agnostic way to define the type for the interval returned by `setInterval`. This complicated looking type: `ReturnType<typeof setInterval>` is basically saying "use the type of whatever setInterval happens to be".

The only other method here is `close` - we can call this from our template to dismiss the modal at any point.

Summary

This was another reasonably simple lesson, but we covered some cool stuff here. Most of the main functionality for the application has been implemented, now we just need to add the bells and whistles. We will, of course, make it look a lot prettier, but in the next lesson, we are going to look at incorporating local notifications.

Lesson 7: Integrating Local Notifications

We've finished creating a lot of the main functionality in the application: we can take photos and have them displayed in a list, delete photos, load photos from storage, play a slideshow and so on. The feature we will be adding in this lesson is more of a nice-to-have that will improve the user experience. We will be adding local notifications that will send a reminder to the user once a day to take their photo for the day.

Let's jump right in.

Local Notifications

Before we implement the local notifications, it might be worth highlighting the difference between **push notifications** and **local notifications**. They both look and behave very similar, they allow you to notify the user of things whether they currently have your application open or not and look like this:

The difference is that push notifications require an external server that handles pushing a notification out to the device, whereas local notifications are completely handled on the device itself. This means local notifications are great for things like alarms, scheduled notifications, or in the case of our application a daily reminder to take a photo. Push notifications are better for being notified of things that happen outside the user's device, like someone sending them a message on Facebook. To use local notifications, we need to know ahead of time when the notification should occur when the user is using the application.

IMPORTANT: The local notification functionality currently has issues, and can not reliably be depended upon. Notification scheduling seems to be impacted by whether the application is open or not, and notifications that have been schedule do not always show up. Whilst you might still want to use this as a supplemental feature, if you require

dependable notifications I would use push notifications instead (which unfortunately is a lot more complex to set up).

To implement this functionality we are just going to add a little bit of code to our root component.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { PhotoService } from "./services/photo.service";

const { SplashScreen, StatusBar, LocalNotifications } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(private photoService: PhotoService) {
    this.initializeApp();
  }

  async initializeApp() {
    await this.photoService.load();

    SplashScreen.hide().catch((err) => {
```

```
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#eb445a" }).catch((err)
=> {
      console.warn(err);
    });

    const pendingNotification = await
LocalNotifications.getPending();

    if (pendingNotification.notifications.length === 0) {
      const firstNotificationTime = new Date();

firstNotificationTime.setHours(firstNotificationTime.getHours() +
24);

      LocalNotifications.schedule({
        notifications: [
          {
            id: 1,
            title: "Snapaday",
            body: "Have you taken your snap today?",
            schedule: {
              at: firstNotificationTime,
              every: "day",
            },
          },
        ],
```

```
      });
    }
  }
}
```

We can access the functionality for local notifications through `LocalNotifications` from Capacitor. We first check if there are already pending notifications, and if there aren't any we create a new notification.

We simply supply a title and add a message that we want to display on the notification, and we also supply an `id` so that we can identify the notification later if needed. We need to describe when to display the notification as well, so we create a time that is 24 hours from now (so that the first notification displays at the same time tomorrow) and set the frequency to every day, so that it will continue to show every day (there are other options you can use like `minute`, `hour`, `week`, and so on).

That's all there is to using this plugin, or at least that's all we need to do with it, there are a lot more options so make sure to check out the documentation for the plugin if you're looking to do something more complex.

Summary

Although this feature is simple, it is likely to increase user engagement in the application given the purpose of the application. Even if someone wanted to take a selfie every day through the application, they would likely forget without some kind of reminder.

The only thing we have left to do now is style our application so that it doesn't look so ugly, we will be doing that in the next lesson.

Lesson 8: Styling

We're almost there, the application is basically completed now, but after this lesson, the functionality will be completed *and* it will look pretty too.

If you remember from the basics section, there are quite a few different ways we can add styles to the application. If you skipped over that part or are not entirely sure what I'm talking about here, I'd recommend going back and reading about creating themes/styling in the basics sections.

> Add the following rule to the bottom of the `global.scss` file:

```
ion-content {
  --ion-background-color: #222222;
}
```

This will modify the `--ion-background-color` CSS variable globally so that all of our backgrounds are this dark colour. We only have two pages in this application so it's not really a huge deal, but you can see how something like this can come in quite useful for making sweeping changes across a larger application.

Now let's move on to the component specific styles.

> Modify `src/app/home/home.page.scss` to reflect the following:

```
ion-list {
  padding: 0;
}

ion-title img {
  max-height: 30px;
  margin-top: 4px;
}

ion-item {
  margin-bottom: 2px;
  --inner-padding-end: 0px;
  --padding-start: 0px;

  img {
    width: 100%;
    height: auto;
  }
}

ion-badge {
  position: absolute;
  right: 10px;
  top: 10px;
}
```

There's nothing too exciting here - we're making sure our photo takes up all the available width and using some absolute positioning to get our "days ago" badge to display where we want it. Most of it is just standard CSS properties, but notice that we are also doing this:

```
--inner-padding-end: 0px;
--padding-start: 0px;
```

You don't just have to set CSS variables globally, in this case, we have changed these variables for anything that matches the `ion-item` selector (this removes the left and right padding on the items). Now let's take a look at the slideshow page.

> Modify `src/app/slideshow/slideshow.page.scss` to reflect the following:

```
.image-container {
  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  display: flex;
  justify-content: center;
  align-items: center;

  img {
```

```
    width: 100%;
    height: auto;
    vertical-align: middle;
  }
}
```

The only styling we're adding to this page is to get the photos to display in the center, both horizontally and vertically (which isn't always the easiest task!). We've done a bit of trickery with `flex` to get that to happen. Flexbox is a powerful way to control layouts with CSS, but it's a bit out of scope for this book. If you're looking for a bit of an intro to it then take a look at [this tutorial].

If you take a look at your application now, it should look something like this (if you've already taken a photo or if you're using the test data):

1 days ago

As you can see, we haven't even added that much styling to the application but it looks a whole lot better now.

Conclusion

Congratulations on making it through the Snapaday application. We've learned a lot through developing this application, but the main points are:

- How to integrate native plugins
- How to use the Camera API
- How to use the File API
- How to use local notifications
- How to use modals
- How to create a custom service
- How to store data permanently

There's always room to take things further though, especially when you're trying to learn something. Following tutorials is great, but it's even better when you figure something out for yourself. Hopefully, you have enough background knowledge now to start trying to extend the functionality of the application by yourself, here are a few ideas to try out:

- Create a different theme for the application **[EASY]**
- Modify the 'Days Ago' pipe to display 'today' and '1 day ago' instead of '0 days ago' and '1 days ago' **[MEDIUM]**
- Add settings to allow the user to control the playback speed of the slideshow **[HARD]**
- Allow the user to configure whether or not they want daily notifications and what time they want to receive them **[HARD]**

Remember, the Ionic documentation is your best friend when trying to figure things out.

What next?

You have a completed application now, but that's not the end of the story. You also need to get it running on a real device and submitted to app stores, which is no easy task. The final sections in this book will walk through how to take what you have done here, and get it onto the app stores so make sure to give that a read.

Camper Mate

Lesson 1: Camper Mate Introduction

Camper Mate is an interesting application to build because it doesn't really have one specific purpose like the other applications have had, it's a bit of a utility toolbelt kind of app. It provides a bunch of different features to users that might be useful for them when going on caravan or camping trips.

This gives us a chance to play around with some things that we haven't been able to in the other applications, and it also presents some interesting challenges. The two most important concepts we'll be covering in this application are the integration and use of Google Maps and the use of forms for capturing user input.

The exact features in this application will be:

- A map that the user can set their camp location on. When they have left their campsite they will be able to click a button which will then show them how to get back to their campsite.
- A form that will allow the user to store personal details relevant to camping (car and trailer registration etc.)
- A form that will allow the user to store details about the camp they are staying at (access codes, WiFi password, etc.)

and some of the main concepts we will be covering are:

- Creating complex forms and capturing user data
- Implementing Google Maps
- Creating a custom component
- Implementing a tabs layout
- Saving and retrieving multiple sets of data

Here are a few screenshots to give you an idea of what it will look like in the end:

Find your camp
Map
Satellite
UnitingSA Westminster Village Aged Care
Military Rd
Fort St
Charles Sturt Ave
Grange
Grange
Baker St
Grange Jetty Cafe
Seaview Rd
Esplanade
Military Rd
Swan St
Crystal Wave
Military Rd
Google
Map data ©2020
Terms of Use
Location
My Details
Camp Details

CamperMATE
Car Registration
IONCAR
Trailer Registration
TZZX43
Trailer Dimensions
8 x 16ft
Phone Number
5555555
Notes
Location
My Details
Camp Details

Lesson Structure

1. Getting Ready
2. Creating a Tabs Layout
3. User Input and Forms
4. Implementing Google Maps and Geolocation
5. Saving and Retrieving Data
6. Styling

Ready?

Now that you know what you're in for, let's get to building it!

Lesson 2: Camper Mate Getting Ready

In this lesson, we are going to prepare our application for the journey ahead. We are going to generate the application, and we are also going to set up all of the components, plugins, and routes that we need. The idea is to get all of the general scaffolding required for most projects out of the way so we don't have to mess around with creating files and configuring things throughout the rest of the lessons.

At the end of this first lesson, we should have a nice skeleton application set up with everything we need to start diving into coding.

A good rule of thumb before starting any new application is to make sure you have the latest version of the Ionic CLI installed, so if you haven't done it recently then make sure to run:

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

If your environment is not set up correctly, you may run into permission errors when installing this package globally. If you are having permission issues, you can use the Ionic teams guide to solving these issues: Resolving permission errors.

If you do not already have a recent version of Node/npm set up on your machine, make sure to complete the Environment Setup guide first.

Generate a new application

We will be using the blank starter template for this application which, as the name implies, is basically an empty Ionic project. It comes with one page built in called **HomePage** but we will be adding some additional pages and making some modifications.

> Run the following command to generate a new application

```
ionic start campermate blank --type angular
```

When you are prompted with the following:

```
? Integrate your new app with Capacitor to target native iOS and
Android? (y/N)
```

Say **Yes**, as we will be utilising Capacitor for native builds. If you do not set up Capacitor when first creating your application, you can always do it manually later as well by running the following command:

```
ionic integrations enable capacitor
```

NOTE: You should still install Capacitor in this project even if you do not intend to build for iOS or Android, as we will still be using Capacitor for the web platform as well.

> Make the new project your current working directory by running the following command:

```
cd campermate
```

Your project should now be generated - now you can open up the project folder in your favourite editor. You can take a look at how your application looks by running the following command:

```
ionic serve
```

which for now should look something like this:

Blank

Ready to create an app?

Start with Ionic UI Components

Create the Required Components

We're going to be creating a few pages for this application, we are going to reuse the automatically generated home page to create our tab layout (which will allow us to switch between pages), but we still need to add pages for the location, my details and camp details tab.

NOTE: You will need to stop serving your application in order to run these commands. If your application is currently being served through the command line, make sure to hit `Ctrl + C` first to stop the process.

> Run the following command to generate the CampDetails page:

```
ionic g page CampDetails
```

> Run the following command to generate the MyDetails page:

```
ionic g page MyDetails
```

> Run the following command to generate the Location page:

```
ionic g page Location
```

In this application, we will actually be creating our own custom component to implement Google Maps. This is really no different to a normal page - each page we create is just its own custom component - it's mostly just that we want to differentiate between our page components which act as our "views" and the components that we use within our pages.

> Run the following command to generate the GoogleMap component:

```
ionic g component components/GoogleMap
```

NOTE: Notice that we use `components/GoogleMap` instead of just `GoogleMap` this time. This is because we want to generate all of our components inside of a folder called `components`.

When we are working with the Google Maps JavaScript SDK we will need to make sure we have the appropriate types installed so that the TypeScript compiler knows what `google` is.

Create the Required Services

As well as our tab pages, we are also going to create a data service to handle saving and retrieving data for us.

> Run the following command to generate a Data service:

```
ionic g service services/Data
```

Create the Interfaces

We will eventually be defining custom interfaces to represent the data that we want to display in the application and save to storage. We will be making use of `Location` data to mark a position on the map, `CampDetails` to represent the data object related to camp details, and `MyDetails` to represent the data object related to personal details. There is

no generate command for this, so you will need to create it manually.

> Create a new folder inside of the `app` folder called `interfaces`

> Create a new file inside of the interfaces folder called `location.ts`

> Create a new file inside of the interfaces folder called `camp-details.ts`

> Create a new file inside of the interfaces folder called `my-details.ts`

Configure the Routes

As we discussed in the basics section, we need to define routes so that our application can match up the current URL to a particular component that you want to display. Generating the application and using the generate commands do most of the work for us, as routes are automatically injected into the **app-routing.module.ts** file. However, we will need to make some modifications to these.

The routes in this application are going to work a little differently than what you would be used to if you have completed the previous examples. This time we will be making use of nested routes, where the home page is going to define its own additional `children` routes since it is acting as the host for the tabs that will contain the other pages. We talk through this concept more in the next lesson, for now, let's just get everything set up.

> Modify `src/app/app-routing.module.ts` to reflect the following:

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

const routes: Routes = [
  {
    path: "",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Even though we have multiple pages in our application, we are only defining the default route here. The home page will host our tabs, and we will define additional routes in the home page module (to handle navigating between individual tabs).

Now let's create the routing for the tabs that will live on our home page.

> Create a file at `src/app/home/home-routing.module.ts` and add the following:

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";
import { HomePage } from "./home.page";

const routes: Routes = [
  {
    path: "tabs",
    component: HomePage,
    children: [
      {
        path: "location",
        children: [
          {
            path: "",
            loadChildren: () =>
              import("../location/location.module").then((m) =>
m.LocationPageModule),
          },
        ],
      },
      {
        path: "camp",
        children: [
          {
            path: "",
            loadChildren: () =>
              import("../camp-details/camp-
```

```
details.module").then((m) => m.CampDetailsPageModule),
          },
        ],
      },
      {
        path: "me",
        children: [
          {
            path: "",
            loadChildren: () =>
              import("../my-details/my-details.module").then((m)
=> m.MyDetailsPageModule),
          },
        ],
      },
    ],
  },
  {
    path: "",
    redirectTo: "/tabs/location",
    pathMatch: "full",
  },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
```

```
export class HomePageRoutingModule {}
```

Now we have set up each of our pages that will be displayed as tabs inside of the home page, as child routes of the home page. Again, we will be discussing how this works in more detail in the next lesson.

We will also be making use of the `ReactiveFormsModule` in both our `CampDetailsPage` and the `MyDetailsPage`. So, we will need to replace the `FormsModule` with `ReactiveFormsModule` in the module files for both of these pages.

> Modify `src/app/camp-details/camp-details.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { ReactiveFormsModule } from "@angular/forms";

import { IonicModule } from "@ionic/angular";

import { CampDetailsPageRoutingModule } from "./camp-details-
routing.module";

import { CampDetailsPage } from "./camp-details.page";

@NgModule({
  imports: [CommonModule, ReactiveFormsModule, IonicModule,
```

```
CampDetailsPageRoutingModule],
  declarations: [CampDetailsPage],
})
export class CampDetailsPageModule {}
```

> **Modify `src/app/my-details/my-details.module.ts` to reflect the following:**

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { ReactiveFormsModule } from "@angular/forms";

import { IonicModule } from "@ionic/angular";

import { MyDetailsPageRoutingModule } from "./my-details-
routing.module";

import { MyDetailsPage } from "./my-details.page";

@NgModule({
  imports: [CommonModule, ReactiveFormsModule, IonicModule,
MyDetailsPageRoutingModule],
  declarations: [MyDetailsPage],
})
export class MyDetailsPageModule {}
```

Set up Additional Modules

Since we are using our custom Google Maps component on the location page, we will need to declare the component in the module for that page.

> Modify `src/app/location/location.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";

import { IonicModule } from "@ionic/angular";

import { LocationPageRoutingModule } from "./location-
routing.module";

import { LocationPage } from "./location.page";
import { GoogleMapComponent } from "../components/google-
map/google-map.component";

@NgModule({
  imports: [CommonModule, FormsModule, IonicModule,
LocationPageRoutingModule],
  declarations: [LocationPage, GoogleMapComponent],
})
export class LocationPageModule {}
```

Configure Capacitor

At this point, you should already have Capacitor added to your project. However, if you did not respond **yes** to the prompt about installing Capacitor when initially creating the project, you should manually set it up now.

> Run the following command (if required) to enable Capacitor:

```
ionic integrations enable capacitor
```

Keep in mind that by default your application will be given a **Bundle ID** of `io.ionic.starter`. This Bundle ID is used to identify your native iOS/Android builds, and you will eventually want to change this to something unique to you or your company. You can change it later if you wish, but it is easier to do before we add the iOS and Android platforms. You may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`. If you don't intend on eventually modifying and publishing this project, then this step doesn't matter.

If you look at the **capacitor.config.json** file, you will notice that the `webDir` is listed as `www`. This is the folder that Capacitor will look to when it is copying over the code for your application into the native projects. It doesn't care about the rest of your source code, it only cares about the built output (i.e. the code that is actually run through the browser). By default, the `www` folder will not exist in your project until you perform a build of your application manually. We are going to do that now.

> Run the following command to generate a build of your application:

```
ionic build
```

It is good to use the standard `ionic build` command whenever you want to test your application on a device during development, but keep in mind that this creates a development build. When you are building the final version of your application, or if you want to test a production version of your application, you should run:

```
ionic build --prod
```

which will create an optimised production build.

Add Native Platforms

By default, the iOS and Android platforms will not be enabled in your project. If you want to add these platforms you can do so now (or you can do it later if you prefer). Remember, building for iOS will require Xcode and building for Android will require Android Studio - if you want to add these platforms now, you should make sure you have followed the instructions in the Generating an Ionic Application lesson to install Xcode, Android Studio, and their dependencies.

```
ionic cap add ios
```

Keep in mind that the first time the `Updating iOS native dependencies` line is run it may take quite a while.

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your Ionic application code and dependencies at any time by running:

```
ionic cap sync
```

IMPORTANT: If you want to use a production build with Capacitor, you will need to add the `--prod` flag to the sync command, e.g: `ionic cap sync --prod`.

When you make changes to your application build, you will need to run this command to copy the changes over to the native iOS/Android projects created by Capacitor. If you do not run this command, any changes you make to your Ionic application will not be present when running the application on iOS/Android.

Set up Root Component

The default starter templates for Ionic utilise **Ionic Native** to handle hiding the **Splash Screen** and styling the **Status Bar**. Whilst you can still use **Ionic Native** and **Cordova** plugins inside of a Capacitor project, Capacitor provides this functionality through default

APIs, so we are going to use those instead. It is also important to remember that the Cordova Splash Screen plugin is not compatible with Capacitor, so make sure that you do not install it in your project.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor() {
    this.initializeApp();
  }

  async initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#54a0ff" }).catch((err)
=> {
```

```
      console.warn(err);
    });
  }
}
```

You can also use the `StatusBar` API to hide the status bar entirely, but in this case we are styling it with a colour that will match our application. The status bar is the bar at the top of native applications that display the time, battery level, etc.

Set up Ionic Storage

We will be making use of the Ionic Storage API in this application. Ionic provides a simple key/value storage API that we can use in our applications to store data - it provides a consistent API and will automatically use the best storage mechanism available. That means that if we install the SQLite plugin in our project, it will store our data in native storage (rather than browser local storage which is more likely to get wiped by the operating system randomly). If native storage is not available, it will default back to browser storage.

To enable Ionic storage, you need to install it.

> Install Ionic Storage with the following command:

```
npm install @ionic/storage
```

and you will also need to add it to your root module file.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported `IonicStorageModule` and we have also added it to the

`imports` array. We have also removed the `SplashScreen` and `StatusBar` providers from Ionic Native since we are not using them anymore.

Set up Plugins

We are going to set up any native plugins that we need now. Remember that when installing Cordova plugins in a Capacitor project you should install them using the `npm install` command, **not** with `ionic cordova plugin add`.

> Run the following command to add the SQLite plugin:

```
npm install cordova-sqlite-storage
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

Remember, after making changes to the native plugins in your project you will need to run:

```
ionic cap sync
```

This will copy the new plugins you have installed over to the native projects.

Set up Images

When building this application we are going to be making use of a few images. I've included these in your download pack but you will need to set them up in the application you generate.

> Copy the images folder in the download pack for this application from `src/assets` to your own `src/assets` folder

Summary

That's it! We're all set up and ready to go, now we can start working on the interesting stuff. Make sure to test your application by running `ionic serve` every now and then, it's easier to catch and fix errors along the way than waiting until later. If you ever get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again. Please note that in the case of this application, our tabs are not going to display properly until we implement them in the next lesson.

Lesson 3: Creating a Tabs Layout

In this lesson, we are going to create the basic layout for the application, which will be a tabs layout. A tabs page works a little differently to most other pages you would have created. To implement a tabs layout we will have one page/component (typically the "home" or a dedicated "tabs" page) that acts as a sort of "host" for the other tabs. The actual content of the tabs are other pages that you import into the "host" page. Once you see it in action it'll probably make more sense.

Let's start off with our Home page which will be the page that holds all of our tabs.

> Modify `src/app/home/home.page.html` to reflect the following

```
<ion-tabs>
  <ion-tab-bar color="primary" slot="bottom">
    <ion-tab-button tab="location">
      <ion-icon name="navigate"></ion-icon>
      <ion-label>Location</ion-label>
    </ion-tab-button>

    <ion-tab-button tab="me">
      <ion-icon name="person"></ion-icon>
      <ion-label>My Details</ion-label>
    </ion-tab-button>
```

```
    <ion-tab-button tab="camp">
      <ion-icon name="bookmarks"></ion-icon>
      <ion-label>Camp Details</ion-label>
    </ion-tab-button>
  </ion-tab-bar>
</ion-tabs>
```

All we have to do to set up our tabs layout when using Angular is to create an `<ion-tabs>` and add an `<ion-tab-bar>` to it. The tab bar should specify whether it should be at the `bottom` or `top`, and we also need to define each of the tab buttons we want to display in the tab bar using `<ion-tab-button>`. By giving the `<ion-tab-button>` a `tab` property that matches the path (according to the routes we defined) of the page we want to display, it will automatically handle displaying the correct page in the tabs layout for us. We can then add whatever we like inside of `<ion-tab-button>` to style our button (usually we would just put a nice icon and label in there).

The tabs we have set up are:

- A tab for the Google Maps component
- A tab for the user to record their details
- A tab for the user to record campsite details

We've already set up the routes for our home page in the getting ready lesson, but we are going to talk through it in more depth now because it is a little different to our standard routes that we use.

> Open `src/app/home/home-routing.module.ts`:

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";
import { HomePage } from "./home.page";

const routes: Routes = [
  {
    path: "tabs",
    component: HomePage,
    children: [
      {
        path: "location",
        children: [
          {
            path: "",
            loadChildren: () =>
              import("../location/location.module").then((m) =>
m.LocationPageModule),
          },
        ],
      },
      {
        path: "camp",
        children: [
          {
            path: "",
```

```
            loadChildren: () =>
              import("../camp-details/camp-
details.module").then((m) => m.CampDetailsPageModule),
          },
        ],
      },
      {
        path: "me",
        children: [
          {
            path: "",
            loadChildren: () =>
              import("../my-details/my-details.module").then((m)
=> m.MyDetailsPageModule),
          },
        ],
      },
    ],
  },
  {
    path: "",
    redirectTo: "/tabs/location",
    pathMatch: "full",
  },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
```

```
  exports: [RouterModule],
})
export class HomePageRoutingModule {}
```

This doesn't look too dissimilar to a standard route setup, except that we are listing our tab routes as `children` of the route that loads the `HomePage` component (and those routes, in turn, have their own `children`). When we are displaying a particular tab, you can imagine that tab is kind of in its own little world with its own routing. If we were to take just the `location` tab, for example, its routing would look like this:

```
{
  path: "location",
  children: [
    {
      path: "",
      loadChildren: () =>
        import("../location/location.module").then((m) =>
m.LocationPageModule),
    },
  ],
},
```

As it stands right now, our `location` tab just has a single route (which is the default one). This then uses the `loadChildren` property to lazy load the location module. Although we

are just displaying a single page within our `location` tab, we could add additional routes here if we wanted so that we could navigate to additional pages *within* the `location` tab. Also, take note that the path of `location` matches up to the `tab` property we gave our `<ion-tab-button>` earlier.

Another important point to note here is that rather than just using a blank path for our `HomePage` component, we are using a `path` of `tabs`:

```
const routes: Routes = [
  {
    path: "tabs",
    component: HomePage,
    ...
```

...and then we redirect the default path to that using:

```
  path: "",
  redirectTo: "/tabs/location",
```

The reason for this is that if we were to attempt to just use the default path and load up the component, the component wouldn't know what tab to display as we haven't specified a particular tab (the home page itself doesn't really display anything, the specific tabs within it do). With the redirect, we can activate a specific tab, and you can use this to specify the

tab that should be loaded by default.

Our tabs layout is basically set up now (hooray!), it's not break time yet though - we're also going to set up the layouts and classes for each of the three tabs.

Interfaces

Before we get to implementing our layouts and classes, let's first define the three interfaces we created earlier. As I mentioned, these are `Location`, `CampDetails`, and `MyDetails`. Each of these will be objects containing various data, and to ensure type safety and provide autocompletion hints we are going to define what exactly those objects should contain (rather than just allowing them to be arbitrary objects with any kind of data).

> Modify `src/app/interfaces/location.ts` to reflect the following:

```
export interface Location {
  latitude: number;
  longitude: number;
}
```

> Modify `src/app/interfaces/camp-details.ts` to reflect the following:

```
export interface CampDetails {
  gateAccessCode: string;
```

```
  ammenitiesCode: string;
  wifiPassword: string;
  phoneNumber: string;
  departure: string;
  notes: string;
}
```

> Modify `src/app/interfaces/my-details.ts` to reflect the following:

```
export interface MyDetails {
  carRegistration: string;
  trailerRegistration: string;
  trailerDimensions: string;
  phoneNumber: string;
  notes: string;
}
```

Tab Layouts

Let's start with the Location page. The location page will contain a map as well as some buttons that will allow the user to set their location, and launch directions so that they can get back home.

> Modify `src/app/location/location.page.html` to reflect the following:

```
<ion-header class="ion-no-border">
  <ion-toolbar color="primary">
    <ion-title>Find your camp</ion-title>
    <ion-buttons slot="start">
      <ion-button (click)="confirmSetLocation()">
        <ion-icon slot="icon-only" name="flag"></ion-icon>
      </ion-button>
    </ion-buttons>
    <ion-buttons slot="end">
      <ion-button (click)="takeMeHome()">
        <ion-icon slot="icon-only" name="compass"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>
  <!-- Google Map component will be inserted here later -->
</ion-content>
```

The `<ion-toolbar>` allows us to add a header bar to the top of our application that can hold buttons, titles, and more (chances are that you are familiar with this by now). We add the `primary` colour to the toolbar to style it with our primary colour.

We set up a title for the page and we also use `<ion-buttons>` to create some buttons in the toolbar. By supplying the `start` slot, the buttons will be placed on the left side of the

toolbar, and by supplying the end slot the buttons will be placed on the right side. We place one button in each position to trigger the "set location", and "take me home" features.

Finally, we have the content area which will eventually be responsible for holding our Google Maps component, but we will be building that later.

Now let's take a look at the class definition for the location page.

> Modify `src/app/location/location.page.ts` to reflect the following:

```
import { Component, OnInit, ViewChild } from "@angular/core";
import { Geolocation } from "@capacitor/core";
import { Platform, AlertController, LoadingController } from
"@ionic/angular";
import { GoogleMapComponent } from "../components/google-
map/google-map.component";
import { Location } from "../interfaces/location";
import { DataService } from "../services/data.service";

@Component({
  selector: "app-location",
  templateUrl: "./location.page.html",
  styleUrls: ["./location.page.scss"],
})
export class LocationPage implements OnInit {
  //@ViewChild(GoogleMapComponent, { static: true }) map:
```

```
GoogleMapComponent;
  private location: Location;

  constructor(
    private platform: Platform,
    private dataService: DataService,
    private alertCtrl: AlertController,
    private loadingCtrl: LoadingController
  ) {}

  async ngOnInit() {}

  async confirmSetLocation(): Promise<void> {}

  async setLocation(): Promise<void> {}

  takeMeHome(): void {}
}
```

We've set up quite a few things here but not much is going on yet. We're setting up a reference to `Geolocation` from Capacitor which we will use later, as well as our Data service (which we will create later as well), and a few APIs from the `@ionic/angular` package.

We inject all of the services we require in the constructor and set up references to them throughout the class by adding the `private` keyword. Finally, we have added the two methods that the buttons in the template call, and one additional method that will allow the user to confirm their action before updating their current location. We will implement there

functionality of these later.

The weirdest thing here is the use of **@ViewChild**, which is currently commented out. You can use `@ViewChild` to grab a reference to an element in the template. So, if we do this:

`@ViewChild(GoogleMapComponent, {static: true}) map: GoogleMapComponent;`

Angular will look in the **location.page.html** template for an element with a selector that matches the `GoogleMapComponent` component. If it finds it, it will return a reference to that component which will be stored under `this.map`. Of course, we haven't created this component yet which is why we have commented it out for now. However, later, we will use this reference to call public methods *inside* of our custom Google Maps component.

You can also use `@ViewChild` to grab a reference to other components. For example, if you wanted to grab a reference to the `<ion-content>` component you could do so using:

```
@ViewChild(Content, {static: true}) contentArea: Content;
```

As long as you import `Content` from the `@ionic/angular` package. Now let's move on to the Camp Details page.

> Modify `src/app/camp-details/camp-details.page.html` to reflect the following:

```
<ion-header class="ion-no-border">
  <ion-toolbar color="primary">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-list lines="none">
      <ion-item>
        <ion-label position="stacked">Gate Access Code</ion-label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Ammenities Code</ion-label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">WiFi Password</ion-label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Phone Number</ion-label>
        <ion-input clearInput type="number"></ion-input>
```

```
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Departure Date</ion-label>
        <ion-datetime displayFormat="DD/MM/YYYY"></ion-datetime>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Notes</ion-label>
        <ion-textarea
          [autoGrow]="true"
          rows="10"
          type="text"
        ></ion-textarea>
      </ion-item>
  </ion-list>
</ion-content>
```

The first thing we do is create the toolbar, but you already know how that works. Then we have a bunch of inputs inside of a list. Rather than using standard HTML like:

```
<input type="text" />
```

We instead use `<ion-input>` which Ionic provides, but it still allows for all the same types (and you will also notice the last field is actually an `<ion-textarea>`). The main

difference between `<ion-input type="text">` and `<input type="text">` is that the Ionic component wraps the standard input and adds its own styling and functionality to it. Notice that we have also added the `clearInput` attribute, which will display a little icon that allows the user to delete the current value in the input.

We supply the `stacked` position which gives us "stacked" labels on our input fields (i.e. the labels will appear above the inputs), but there are a whole bunch of different styles Ionic provides for inputs by default. For now we've just added labels to the fields, but later on, we will come back and add some extra stuff so that we can actually grab the values that a user inputs.

Also notice the use `<ion-datetime>` here, which is an in-built date and time picker that Ionic provides.

There's not going to be much happening yet, but let's also create the class for this page now.

> Modify `src/app/camp-details/camp-details.page.ts` to reflect the following:

```
import { Component, OnInit } from "@angular/core";
import { FormBuilder, FormGroup } from "@angular/forms";
import { DataService } from "../services/data.service";
import { debounceTime } from "rxjs/operators";

@Component({
  selector: "app-camp-details",
  templateUrl: "./camp-details.page.html",
```

```
  styleUrls: ["./camp-details.page.scss"]
})
export class CampDetailsPage implements OnInit {
  public campDetailsForm: FormGroup;

  constructor(formBuilder: FormBuilder, private dataService:
DataService) {}

  async ngOnInit() {}

}
```

The only thing we're doing here that's out of the ordinary is importing **FormBuilder** from the `@angular/forms` package. Just above I mentioned that we will be editing the inputs we created later to actually do something, and this is where `Form Builder` comes in. Form Builder will allow you to create and manage forms, and `Validators` can be attached to specific fields to ensure a valid value is input (i.e. if it needs to be an email address, phone number, etc.). We won't be making use of `Validators` in this application as it does not need them (which is why we hae not imported `Validators` from `@angular/forms`) but we will walk through a hypothetical example of validation in the next lesson (e.g. a scenario where we want to make sure that the user has filled in a particular input).

Now let's move on to the My Details page, which is almost exactly the same as this one.

> Modify `src/app/my-details/my-details.page.html` to reflect the following

```
<ion-header class="ion-no-border">
  <ion-toolbar color="primary">
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-list lines="none">
      <ion-item>
        <ion-label position="stacked">Car Registration</ion-label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Trailer Registration</ion-
label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Trailer Dimensions</ion-
label>
        <ion-input clearInput type="text"></ion-input>
      </ion-item>

      <ion-item>
```

```
        <ion-label position="stacked">Phone Number</ion-label>
        <ion-input clearInput type="number"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Notes</ion-label>
        <ion-textarea
          [autoGrow]="true"
          rows="10"
          formControlName="notes"
          type="text"
        ></ion-textarea>
      </ion-item>
  </ion-list>
</ion-content>
```

There's not much need to explain this one because, apart from having different input fields, it is exactly the same as the Camp Details template. Let's move on to the class.

> Modify `src/app/my-details/my-details.page.ts` to reflect the following:

```
import { Component, OnInit } from "@angular/core";
import { FormBuilder, FormGroup } from "@angular/forms";
import { DataService } from "../services/data.service";
import { debounceTime } from "rxjs/operators";
```

```
@Component({
  selector: "app-my-details",
  templateUrl: "./my-details.page.html",
  styleUrls: ["./my-details.page.scss"]
})
export class MyDetailsPage implements OnInit {
  public myDetailsForm: FormGroup;

  constructor(formBuilder: FormBuilder, private dataService:
DataService) {}

  async ngOnInit() {}

}
```

Once again, this is exactly the same as the Camp Details page. Sorry for the boring finish, but that's all we need to do for now for our layout, in the next lesson we're going to take a closer look at `FormBuilder` and get our inputs working properly.

Lesson 4: User Input and Forms

In this lesson, we're going to be looking at the functionality of the **Camp Details** and **My Details** pages, which will involve making our forms functional. If you've completed some of the other applications in this book, or have tried using inputs in Ionic before, you may have used `ngModel` like this:

```
<ion-input type="text" [(ngModel)]="myField"></ion-input>
```

This sets up two-way data binding on this input field, which I discussed in more detail in the basics section, but essentially it ties the value of this input to:

`this.myField`

in the class definition for the page. If you change the value of `this.myField` it will be reflected in the input, and if you change the value of the input it will be reflected in `this.myField`. This is why it is "two-way" - changing either value will affect the other. This is a pretty easy way to go about managing user input - just bind a member variable to the input. However, if you have large and complex forms, binding everything to member variables/objects manually starts to become a bit cumbersome. This is where it becomes easier to use **FormBuilder**.

As well as making the code a bit nicer, by using `FormBuilder` you can manage each of your inputs by directly using "form controls" which provide powerful functionality that you

can hook into (if you check out the Giflist application you will see that we use a control to subscribe to changes on the input field, which allowed us to do all sorts of fancy things).

We can also make use of **Validators** with Form Builder, which allow us to tie a "validation" to a specific input field, which will check the input to see if it is allowed (e.g. a correctly formatted email address).

Let's jump right into implementing it so that you can see how it works. We'll go through implementing this functionality on the Camp Details page and then replicate it on the My Details page.

> Modify the list in `src/app/camp-details/camp-details.page.html` to reflect the following:

```
  <ion-list lines="none">
    <form [formGroup]="campDetailsForm">
      <ion-item>
        <ion-label position="stacked">Gate Access Code</ion-label>
        <ion-input clearInput formControlName="gateAccessCode"
type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Ammenities Code</ion-label>
        <ion-input clearInput formControlName="ammenitiesCode"
type="text"></ion-input>
      </ion-item>
```

```
      <ion-item>
        <ion-label position="stacked">WiFi Password</ion-label>
        <ion-input clearInput formControlName="wifiPassword"
type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Phone Number</ion-label>
        <ion-input clearInput formControlName="phoneNumber"
type="number"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Departure Date</ion-label>
        <ion-datetime formControlName="departure"
displayFormat="DD/MM/YYYY"></ion-datetime>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Notes</ion-label>
        <ion-textarea
          [autoGrow]="true"
          rows="10"
          formControlName="notes"
          type="text"
        ></ion-textarea>
      </ion-item>
    </form>
```

```
  </ion-list>
```

Our inputs are mostly the same except for a couple of important differences. First, we have wrapped the whole thing in a form tag:

```
<form [formGroup]="campDetailsForm">
```

We define the `formGroup` property as `campDetailsForm` which we will use with the `FormBuilder` very shortly. Usually, on a form, you would instead listen for the `(submit)` event to handle the data submission, however we won't be implementing this as a form that is submitted. We just want fields the user can enter data into, and later we are going to make use of an observable that is provided by `FormBuilder` to save the data every time anything new is entered into these fields.

The other important thing we have changed is that we have added `formControlName` to each of our inputs, giving it a name (similar to what we would do with `ngModel`). Again, we will use this with the `FormBuilder` in just a moment.

Now we're going to take a look at the class definition. First, we are going to modify the constructor:

> Modify the constructor in `src/app/camp-details/camp-details.page.ts` to reflect the following:

```
```

```
  public campDetailsForm: FormGroup;

  constructor(formBuilder: FormBuilder, private dataService:
DataService) {
    this.campDetailsForm = formBuilder.group({
      gateAccessCode: [""],
      ammenitiesCode: [""],
      wifiPassword: [""],
      phoneNumber: [""],
      departure: [""],
      notes: [""]
    });
  }
```

Since we defined the `formGroup` as `campDetailsForm` in the template, we can assign a new `FormBuilder` group to it here. To create the group, we supply all of the **formControlName** attributes that we added to the inputs. Notice that we also supply an array that contains an empty string, this is used to initialise the value of the input, for example:

```
gateAccessCode: ['54321']
```

would set the `gateAccessCode` input value to '54321'. You can also supply a validator here if you like, by supplying an additional element to the array:

```
gateAccessCode: ['', Validators.required]
```

this will make the `gateAccessCode` field a required field (you will also need to import `Validators` from `@angular/forms`). By adding a validator, we are able to easily check which of our form controls have satisfied their validation requirements like this:

```
if(this.campDetailsForm.controls.gateAccessCode.valid){
  // do something
}
```

...and we can even check if the entire form is valid (i.e. all validators for all form controls have been satisfied) like this:

```
if(this.campDetailsForm.valid){
  // do something
}
```

Although we won't be diving too deep into advanced validators in this book, there is a lot more you can do than just make a field required. You might also want to:

- Add a minimum length to a field with: `Validators.minLength(10)`
- Add a maximum length to a field with: `Validators.maxLength(30)`
- Match against a regex pattern: `Validators.pattern('[a-zA-Z ]*')`

You can even have multiple validators on a single form control by using the `compose`

method:

```
someField: ['', Validators.compose([Validators.maxLength(30),
Validators.pattern('[a-zA-Z ]*'), Validators.required])]
```

This only scratches the surface of what we might do with just `Validators`, and then there is even more functionality built into `FormBuilder` and form controls, so hopefully you can see the benefits of setting up your forms this way. For simple forms `ngModel` may be the perfect option, but for anything more complex, using `FormBuilder` will quickly become far more efficient.

That's all there is to set up our form, now we just need to implement a way to save the data entered into these fields.

> Modify `ngOnInit` in `src/app/camp-details/camp-details.page.ts` to reflect the following:

```
  async ngOnInit() {
    this.campDetailsForm.valueChanges.pipe(
      debounceTime(1000)
    ).subscribe(() => {
      //this.dataService.setMyDetails(this.campDetailsForm.value);
    });
  }
```

NOTE: We have commented out the call to the data service since we haven't implemented it yet, otherwise it will cause TypeScript to throw errors at us. We will uncomment this later.

We want to save changes to this form every time the values in the fields change. Since we used `FormBuilder` to create our form, we have access to the `valueChanges` observable. This will emit data every time that data in the form controls change. This is a great way for us to trigger a save when data changes. To avoid unnecessary calls, we pipe in the `debounceTime` operator so that the save will only be triggered once the user stops typing for more than 1 second (if the user is currently typing into the field we don't want to trigger unnecessary saves for every single keystroke).

We can grab all the values from our form at any time by using `this.campDetailsForm.value`. This will return an object that contains all of the values, which is exactly how we would eventually like to store them. So, we pass this object of values through to our data service to save (which we haven't implemented yet).

Now we just need to reflect this functionality in our My Details page. Again, it's pretty much exactly the same so I'll just paste the code below rather than explaining it step by step.

> Modify the list in `src/app/my-details/my-details.page.html` to reflect the following:

```
<ion-list lines="none">
  <form [formGroup]="myDetailsForm">
```

```
      <ion-item>
        <ion-label position="stacked">Car Registration</ion-label>
        <ion-input clearInput formControlName="carRegistration"
type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Trailer Registration</ion-
label>
        <ion-input clearInput
formControlName="trailerRegistration" type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Trailer Dimensions</ion-
label>
        <ion-input clearInput formControlName="trailerDimensions"
type="text"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Phone Number</ion-label>
        <ion-input clearInput formControlName="phoneNumber"
type="number"></ion-input>
      </ion-item>

      <ion-item>
        <ion-label position="stacked">Notes</ion-label>
        <ion-textarea
```

```
          [autoGrow]="true"
          rows="10"
          formControlName="notes"
          type="text"
        ></ion-textarea>
      </ion-item>
    </form>
  </ion-list>
```

> Modify the constructor in `src/app/my-details/my-details.page.ts` to reflect the following:

```
  public myDetailsForm: FormGroup;

  constructor(formBuilder: FormBuilder, private dataService:
DataService) {
    this.myDetailsForm = formBuilder.group({
      carRegistration: [""],
      trailerRegistration: [""],
      trailerDimensions: [""],
      phoneNumber: [""],
      notes: [""]
    });
  }
```

> Modify `ngOnInit` in `src/app/my-details/my-details.page.ts` to reflect the following:

```
async ngOnInit() {
  this.myDetailsForm.valueChanges.pipe(
    debounceTime(1000)
  ).subscribe(() => {
    //this.dataService.setMyDetails(this.myDetailsForm.value);
  });
}
```

Forms aren't exactly the most exciting thing in the world (well, for most people at least), but they are a critical component of many mobile applications so it's important to understand how they work. Being able to use `FormBuilder` can make your forms a lot more manageable and powerful, but sometimes a simple `[(ngModel)]` is enough to do the job as well.

In the next lesson we'll be jumping into something a little more fun, and quite a bit more complicated, when we implement Google Maps!

Lesson 5: Implementing Google Maps and Geolocation

Google Maps and mobile apps are a perfect match. The Google Maps API is an awesome piece of tech by itself, but when you couple it with a device that is meant to be mobile, as in not stationary, it opens up a wide range of possibilities. There are a ton of cool apps out there today that utilise Google Maps to do all kinds of things.

Even if maps aren't the core feature of your application, they are often quite useful as supplementary features as well (displaying the location of a business on a map for example).

In this lesson, we will be implementing Google Maps on our Location page. Essentially what we want to do is:

- Display a map
- Allow the user to set their current location on the map
- Display a marker at their last set location
- Enable the user to launch directions to take them back to their set location

Getting the Google Maps SDK set up in an application, and even using it, is pretty easy. You simply load the Google Maps SDK script and then start interacting with the API. It gets a bit more complicated than that though because of one main problem:

What if the user does not have an Internet connection?

It's not unreasonable to make the maps unavailable if the user does not have an Internet

connection, but how do we handle that gracefully? We don't want an error occurring and breaking the application (because the Google Maps SDK hasn't been loaded) or otherwise causing the maps not to work, so we need to consider the following:

- What if the user does not have an Internet connection?
- What if the user does not have an Internet connection initially but does later?
- What if the user does have an Internet connection initially but doesn't later?

To handle all of these scenarios, the solution we want to implement will:

- Wait until a connection is available before loading the Google Maps SDK, rather than straight away
- If the connection becomes unavailable, disable the Google Maps functionality
- If the connection becomes available again, enable the Google Maps functionality

To make our code a little bit cleaner we are going to abstract a lot of this functionality into the custom Google Maps component that we generated earlier. This will also make it easier to reuse the same code in other applications as well.

NOTE: We will be using the Google Maps JavaScript SDK in this lesson, but you should also be aware that you can use Google's native SDK as well through this Cordova plugin: https://github.com/mapsplugin/cordova-plugin-googlemaps. Personally, I think that the Javascript SDK is generally better suited for Ionic applications as they are both web-based, but the native plugin may be better suited to some scenarios. The native plugin also allows for better API key restriction/security as we will discuss later.

This lesson is going to be pretty big so let's jump right into it.

Google Maps Component

The point of this custom component is to hold all of our complex logic related to Google Maps. The end goal is to be able to simply drop an element like this:

```
<google-map apiKey="MY-GOOGLE-MAPS-API-KEY"></google-map>
```

into the template where we want to embed Google Maps, and just about everything will be handled automatically for us. We need to do some hard work up front the first time, but after that, it is extremely easy to reuse.

It's a pretty big component, so we'll create a bit of a skeleton first and then implement it function by function.

> Modify `src/app/components/google-map/google-map.component.ts` to reflect the following:

```
import { Component, AfterViewInit, ElementRef, Renderer2, Input,
Inject } from "@angular/core";
import { DOCUMENT } from "@angular/common";
import { Plugins } from "@capacitor/core";
import { BehaviorSubject, Observable } from "rxjs";
import { takeWhile } from "rxjs/operators";

const { Geolocation, Network } = Plugins;
```

```
@Component({
  selector: "app-google-map",
  templateUrl: "./google-map.component.html",
  styleUrls: ["./google-map.component.scss"],
})
export class GoogleMapComponent implements AfterViewInit {
  @Input("apiKey") apiKey: string;

  public loadedStatus: "initial" | "loading" | "loaded" | "failed"
= "initial";
  private loadedStatus$: BehaviorSubject<
    "initial" | "loading" | "loaded" | "failed"
  > = new BehaviorSubject<"initial" | "loading" | "loaded" |
"failed">(this.loadedStatus);

  public map: any;
  public connectionAvailable: boolean = true;
  private googleMaps: any;
  private marker: any;

  constructor(
    private element: ElementRef,
    private renderer: Renderer2,
    @Inject(DOCUMENT) private _document: Document
  ) {}

  ngAfterViewInit() {
    this.init();
```

```
  }

  public getLoadedStatus(): Observable<"initial" | "loading" |
"loaded" | "failed"> {
    return this.loadedStatus$;
  }

  public async init(): Promise<void> {}

  public changeMarker(lat: number, lng: number): void {}

  private async load(): Promise<void> {}

  private injectSdk(): void {}

  private handleSdkLoaded(): void {}

  private async initMap(): Promise<void> {}

  private addConnectivityListener(): void {}
}
```

To begin with, let's talk through our imports as there are a couple of new things here:

- `Input` - this will allow us to grab the `apiKey` value that is supplied to the component. We grab this value using the `@Input` decorator.
- `Renderer2`, `ElementRef` - The `ElementRef` gives us access to the native DOM element of the component, and `Renderer2` provides us with some functions to

modify the DOM in an Angular friendly manner

- `Inject`, `DOCUMENT` - We are going to be injecting the Google Maps SDK directly into the DOM which requires access to the `document` object. Using `Inject` and `Document` provides us with an Angular friendly way to access the `document` object. This is basically just an Angular way to interact with the standard `document` object available in JavaScript.
- `BehaviorSubject`, `Observable`, `takeWhile` - just like we have with previous applications, we are going to use a `BehaviorSubject` which is a type of `Observable` to provide data updates to the rest of our application. In this case, our `BehaviorSubject` will allow us to indicate the current loading status of the maps. We will us `takeWhile` to automatically unsubscribe from a subscription in our Google Maps component.

We also have a bunch of member variables set up:

- `loadedStatus` will keep track of the current loading state. This can have a value of `initial` which indicates that the process has not yet begun, `loading` which means we are currently attempting to load the maps, `loaded` which means we have successfully finished loading the maps, and `failed` which means we attempted to load the maps but it failed.
- `loadedStatus$` is our `BehaviorSubject` which we will use to expose the `loadedStatus` value to the rest of the application
- `map` will hold a reference to our Google Map
- `connectionAvailable` will indicate whether there is an active network connection available or not
- `googleMaps` will hold a reference to the Google Maps JavaScript SDK

- `marker` will hold a reference to the marker that has been added to the map (indicating the camp location)

Then we have our various methods. We kick off the loading process in our `ngAfterViewInit` hook by calling the `init` method, which starts a chain of method calls that eventually loads the Google Maps JavaScript SDK and initialises a new map. We are going to work through implementing these one at a time.

> Modify the `init` method to reflect the following:

```
  public async init(): Promise<void> {
    switch (this.loadedStatus) {
      case "initial":
        // First attempt at loading
        this.load();
        this.addConnectivityListener();
        break;

      case "failed":
        //Loading previous failed, try again
        this.load();
        break;

      case "loading":
        // A new load has been triggered whilst one is already in
progress
        // Wait to see if the current load fails, if it does try
```

```
again
        this.loadedStatus$
          .pipe(
            takeWhile(
              (loadedStatus) => loadedStatus === "loaded" ||
loadedStatus === "failed",
              true
            )
          )
          .subscribe((loadedStatus) => {
            if (loadedStatus === "failed") {
              this.init();
            }
          });
        break;
      default:
        console.log("Maps already loaded");
    }
  }
```

As we just discussed, there are various "loading states" that this component can be in. We might end up calling this `init` method more than once throughout the life of the component (e.g. if we are attempting to retry a load after an initial failure), so we need to handle all of the different potential loading states.

The first time the `init` method is called the `loadedStatus` will be `initial`. In this case,

we call the `load` method to kick off the loading process and then we add the connectivity listener. We haven't implemented the `addConnectivityListener` method yet, but eventually this will listen for network changes (e.g. if the user goes `online` or goes `offline`) and will trigger some behaviour as a result of that.

If this is not the first time the `init` method has been called, then the `loadedStatus` might be set to `failed` (e.g. we have attempted a load previously but it failed). In this case, we just try the `load` again, but this time there is no need to add the connectivity listener as it has already been added previously.

It is also possible that this `init` method will be called *during* a load. It takes some time for the load to complete, and another `init` might be triggered during this process (e.g. if the user's network status changes during the loading process). In this case, the `loadedStatus` will be `loading`. This is a bit of a tricky one to handle, because we don't want to just ignore the request because it might be that the current load ends up failing, but if we triggered one now it would work. We also don't want to trigger another load whilst one is already in progress, though. To deal with this, we subscribe to `loadedStatus` and wait for the status to either by `loaded` or `failed` (indicating that the load has finished). We use `takeWhile` to automatically unsubscribe from this observable once a `loaded` or `failed` value is received. If the status is now `loaded` we don't need to do anything because the maps have been successfully loaded, but if the status is `failed` we trigger the `init` method again.

Now let's see what our `load` method does.

> Modify the `load` method to reflect the following:

```
  private async load(): Promise<void> {
    this.loadedStatus = "loading";
    this.loadedStatus$.next(this.loadedStatus);

    const win = window as any;

    if (win.google && win.google.maps) {
      // SDK already loaded, skip straight to the next step
      this.handleSdkLoaded();
    } else {
      // Check if the user has a connection before attempting to
load
      // the SDK.
      const networkStatus = await Network.getStatus();

      if (networkStatus.connected) {
        this.injectSdk();
      } else {
        this.connectionAvailable = false;
        this.loadedStatus = "failed";
        this.loadedStatus$.next(this.loadedStatus);
      }
    }
  }
```

First, we update the `loadedStatus` to indicate that a load is now in progress. Once the

Google Maps JavaScript SDK is added to our application, its functionality will be exposed on the global `window` object under `window.google.maps`. TypeScript won't like us trying to access objects it doesn't have type definitions for, so we create our own `win` object using the `any` type. We are operating blind now because we can just access whatever objects we want on `window` and TypeScript will allow it - this is why you generally shouldn't use `any` if its avoidable.

First, we check if the `win.google.maps` object is already available. This shouldn't happen, but maybe the Google Maps JavaScript SDK has already been loaded by some other component in your application. In this case, we just skip straight to the `handleSdkLoaded` method. Otherwise, we check if the user is currently connected to the Internet and if they are we call the `injectSdk` method. If the user is not connected to the Internet, we indicate that the load attempt has failed.

> Modify the `injectSdk` method to reflect the following:

```
private injectSdk(): void {
  window["mapInit"] = () => {
    this.handleSdkLoaded();
  };

  const script = this.renderer.createElement("script");
  script.id = "googleMaps";
  script.async = true;
  script.defer = true;
  script.src = `https://maps.googleapis.com/maps/api/js?
```

```
key=${this.apiKey}&callback=mapInit`;

    this.renderer.appendChild(this._document.body, script);
  }
```

This is the part where we actually inject the Google Maps JavaScript SDK into the document. All we are doing is manually creating a `<script>` element with a `src` of the Google Maps SDK, and then attaching that to the document using the `appendChild` function. Since the Google Maps SDK will trigger a callback function of `mapInit`, we need to define that call back function on the `window` which has global scope. This will allow us to detect when the SDK has finished loading, and then we call our `handleSdkLoaded` method. It is not typical to define any variables or functions on the global `window` object in an Angular application, in fact, you should always avoid doing this, but this is the way that the Google Maps SDK works.

Notice that we are making use of the `DOCUMENT` that we injected earlier, using this in combination with `Renderer2` allows us to inject a new script tag into the body of our application in an Angular way rather than having to just use the standard `document` methods. This makes the code more platform agnostic, but technically either approach would work.

> Modify the `handleSdkLoaded` method to reflect the following:

```
  private handleSdkLoaded(): void {
    const win = window as any;
```

```
    this.googleMaps = win.google.maps;
    this.initMap();
  }
```

This is quite a simple method, all we do is set up a reference to the `google.maps` object on our `googleMaps` member variable. Then, we call `initMap` to create our map instance using the SDK.

> Modify the `initMap` method to reflect the following:

```
  private async initMap(): Promise<void> {

    try {
      const position = await Geolocation.getCurrentPosition({
        enableHighAccuracy: true,
        timeout: 10000,
      });

      const latLng = new this.googleMaps.LatLng(
        position.coords.latitude,
        position.coords.longitude
      );
      const mapOptions = {
        center: latLng,
        zoom: 15,
      };
```

```
      this.map = new
this.googleMaps.Map(this.element.nativeElement, mapOptions);
      this.loadedStatus = "loaded";
      this.loadedStatus$.next(this.loadedStatus);
    } catch (err) {
      console.log(err);
      this.loadedStatus = "failed";
      this.loadedStatus$.next(this.loadedStatus);
    }
  }
```

We want to initialise our map using the user's current location as the center, so we first make a call to our `Geolocation` plugin to grab the users current position. This will return us the `latitude` and `longitude` of wherever the user is currently. We then use that to create a new Google Maps `LatLng` object, which we can then supply to the SDK when creating a new map. To create a map, we just create a `new` `google.maps.Map` object passing in the element we want to attach the map to, and our options including the user's current location.

Once this is done, we indicate that the load has finished. If this fails, we indicate that the load has failed. We are mostly done now, but there are a couple more methods for us to finish implementing.

> Modify the `addConnectivityListener` method to reflect the following:

```
private addConnectivityListener(): void {
  Network.addListener("networkStatusChange", (status) => {
    if (status.connected) {
      if (this.loadedStatus !== "loaded") {
        // Attempt to reload the SDK
        this.init();
      } else {
        this.connectionAvailable = true;
      }
    } else {
      this.connectionAvailable = false;
    }
  });
}
```

We are using the `Network` API from Capacitor here to listen for whenever the user's Internet connection changes. We keep track of the current connection status with our `connectionAvailable` boolean, and if the user has come online and the `loadedStatus` is not already `loaded` then we trigger `init` again.

> Modify the `changeMarker` method to reflect the following:

```
public changeMarker(lat: number, lng: number): void {

  const latLng = new this.googleMaps.LatLng(lat, lng);
```

```
    const marker = new this.googleMaps.Marker({
      map: this.map,
      animation: this.googleMaps.Animation.DROP,
      position: latLng,
    });

    // Remove existing marker if it exists
    if (this.marker) {
      this.marker.setMap(null);
    }

    // Add new marker
    this.marker = marker;
  }
```

The last method we need to implement allows us to add/change the current marker on the map. We want to allow the user to set the location of their camp. When this method is triggered it will take in a `latitude` and `longitude` and then add a new marker on the map at that location. We only want one marker on the map at a time, so if there is already an existing marker on the map we remove it.

NOTE: Notice how the `changeMarker` method is marked as `public` whereas most others are marked as `private`? This means that we will be able to call this `changeMarker` method from outside of this component - we will just need a reference to the component.

We've handled all the difficult logic for our component, but we still need to implement the template and a little bit of styling so that it all works (fortunately, this will very easy).

> Modify `src/app/components/google-map/google-map.component.html` to reflect the following:

```
<div id="connection-overlay" *ngIf="!connectionAvailable">
  <p>Connection required to use maps...</p>
</div>

<ion-spinner [paused]="loadedStatus === 'loaded'"></ion-spinner>
```

The Google Map will be attached to the component itself, but by adding an `<ion-spinner>` inside of the component, the user will see the spinner as the map is loading. The point of the connection overlay is to indicate the network status to the user, and we will style this so that it will overlay the map when displayed. Notice that we use `*ngIf` so that this element is only added when the `connectionAvailable` flag is `false`.

> Modify `src/app/components/google-map/google-map.component.scss` to reflect the following:

```
:host {
  position: absolute !important;
  display: flex;
  align-items: center;
```

```
  justify-content: center;
  width: 100%;
  height: 100%;
  background-color: var(--ion-color-primary);
}

#connection-overlay {
  position: absolute;
  background-color: #000;
  opacity: 0.5;
  width: 100%;
  height: 100%;
  z-index: 1;
}

#connection-overlay p {
  color: #fff;
  font-weight: bold;
  text-align: center;
  font-size: 1.6em;
}
```

These are the styles for our component, and we need to do a bit of trickery to get the map to display correctly. To modify the styles of the component itself we can use the `:host` pseudo-selector, which we use to give the map absolute positioning and to center everything vertically and horizontally inside of the component with `flex`. We also add some styling for our connection overlay.

Now that we have our Google Maps component completely set up, we just have to use it!

Implementing Google Maps

We've done quite a bit of work to get maps working already, but we still have a little way to go. Now we are going to modify our Location page to make use of the Google Maps component.

Adding the component itself to the template is extremely easy, all we need to do is add it to the content area.

> Modify `<ion-content>` in `src/app/location/location.page.html` to reflect the following:

```
<ion-content>
    <app-google-map apiKey="YOUR-API-KEY"></app-google-map>
</ion-content>
```

Before Google Maps will work, you will need to supply the component with an API key. You can generate an API key with Google using this link. Unfortunately, Google has made recent changes that require adding payment options in order to generate a key. They do still provide a free tier by providing you with free credits and only bill you after exceeding that, but you will need to keep an eye on your usage to avoid any unexpected surprises. According to Google, most users will never exceed the free tier, but you should always be careful. Do keep in mind that the key you use for Google Maps is exposed publicly in your code, so although it may be unlikely to happen, someone could potentially find and use

your API key without your knowledge and add to your usage.

If you are deploying your application as a PWA, then you can restrict the usage of your API key when you create it using an HTTP Referrer set to the domain your application is hosted on. However, if you are deploying the application as a native application on iOS or Android this method will not work (as the request to load Google Maps comes from the embedded browser which will have an HTTP referrer or `localhost`). Google also provides an option to restrict key usage on iOS and Android applications, but this is only for the native Google Maps SDK.

NOTE: You can also restrict your key to only be usable with the `Maps JavaScript API`. By doing this, if your key is used inappropriately, it will only be able to be used to access the Maps API.

There are some creative ways to help protect your Google API key like:

- Using a proxy server you control to handle adding the API key to your requests (e.g. have requests to `https://serveryoucontrol.com/maps/api/js` resolve to `https://maps.googleapis.com/maps/api/js?key=YOUR-API-KEY`)
- You can use a custom URL scheme for Capacitor and restrict it to only be accessible through that URL scheme
- If only authenticated users can use your application, then you could make the key only retrievable for authenticated users

But there isn't really any fool proof method. It is unlikely that someone will steal and use your key to access Google Maps, but it is always possible. To protect against this scenario, I would recommend going to the quotas section and choosing the `Maps`

`JavaScript API`. From here you will find a `Map loads` card that can be expanded. If you expand that and scroll down to the bottom, you should see something like:

```
Map loads per day: Unlimited
Map loads per minutes: 30,000
Map loads per user: 60
```

You can edit these values to limits you are comfortable with. That way, if you do get an unexpectedly high amount of usage on your API key (either genuinely or because of a nefarious person) you can limit how much you would be liable to pay. You could even set these limits low enough such that you would never exceed the amount of free usage available. What exactly this is, is a little bit confusing due to Google's pricing structure - there is (at the time of this writing) no "free usage tier" but you do get a certain amount of free credit each month to use, so you can make use of the maps API without needing to pay. I would recommend setting all of these quota limits as low as you can comfortably can whilst being able to support the number of users you have. There is also a graph available on this page so that you can view any quota exceeded errors that have been triggered (indicating that you would need to increase your quotas).

Once you have your API key set up and supplied to the `<google-map>` component, we can move on. Next, we need to add a way to trigger the `init` method in our component. We already set up a reference to the component using `@ViewChild`, so we just need to uncomment it now.

> Uncomment `@ViewChild` in `src/app/location/location.page.ts`:

```
@ViewChild(GoogleMapComponent, {static: true}) map:
GoogleMapComponent;
```

At this point, our map should be able to load and display the map on our location page, but there are just a couple more things we need to do to finish it off. We are going to implement the `setLocation` method, which will set the user's camp location to their current location.

> Modify the `setLocation` method in `src/app/location/location.page.ts` to reflect the following:

```
  async setLocation(): Promise<void> {
    const overlay = await this.loadingCtrl.create({
      message: "Setting current location...",
    });

    overlay.present();

    try {
      const newLocation = await Geolocation.getCurrentPosition({
        enableHighAccuracy: true,
        timeout: 10000,
      });

      this.location = {
```

```
        latitude: newLocation.coords.latitude,
        longitude: newLocation.coords.longitude,
      };

      this.map.changeMarker(this.location.latitude,
this.location.longitude);
      //this.dataService.setLocation(this.location);

      const alert = await this.alertCtrl.create({
        header: "Location set!",
        message:
          "You can now find your way back to your camp site from
anywhere by clicking the button in the top right corner.",
        buttons: [
          {
            text: "Ok",
          },
        ],
      });

      overlay.dismiss();
      alert.present();
    } catch (err) {
      console.log(err);
      overlay.dismiss();
    }
  }
```

Since it is going to take a few seconds to grab the users location, we pop up a loading overlay to indicate that something is happening to the user. If we didn't do this, then the user would click the button and nothing would happen for a couple of seconds, which isn't a great user experience.

Once again, we are using the Geolocation plugin to grab the user's current position. Once we get that we change the `location` value to the user's current location, and we also call the `changeMarker` function with that position. We want this location to be remembered when the user opens the application again so we create a data object for the position and send it off to our data service to be saved (remember, we haven't actually implemented that yet).

Once this process is complete, we trigger an alert to let the user know that their location was successfully set and what that means. The method above isn't what is actually called when the user taps the button in the toolbar. We first call a `confirmSetLocation` method to confirm that the user wants to perform this action. Let's implement that now.

> Modify the `confirmSetLocation` method in `src/app/location/location.page.ts` to reflect the following:

```
  async confirmSetLocation(): Promise<void> {

    const alert = await this.alertCtrl.create({
      header: "Set new location?",
      message:
        "This will set the location of your camp to wherever you
are currently positioned. Are you sure?",
```

```
    buttons: [
      {
        text: "Cancel",
        role: 'cancel'
      },
      {
        text: "Confirm",
        handler: () => {
          this.setLocation();
        }
      },
    ],
  });

  alert.present();

}
```

All this does it pop up an alert getting the user to confirm, and then we call the `setLocation` method that we just defined. Now we just have one more function to define:

> Modify the `takeMeHome` method in `src/app/location/location.page.ts` to reflect the following:

```
takeMeHome(): void {
```

```
    if (!this.location) {
      const alert = this.alertCtrl.create({
        header: "Nowhere to go!",
        message: "You need to set your camp location first.",
        buttons: [
          {
            text: "Ok",
          },
        ],
      });
    } else {
      const destination = `${this.location.latitude},
${this.location.longitude}`;

      if (this.platform.is("ios")) {
        window.open(`maps://?q=${destination}`, "_system");
      } else {
        const label = encodeURI("My Campsite");
        window.open(`geo:0,0?q=${destination}(${label})`,
"_system");
      }
    }
  }
```

The goal of this function is to show the user directions from their current location to their campsites location. First, we check that the `this.location` value is set, and if it is not we simply display an alert letting the user know they need to set their location first.

If they do exist then we use the Google Maps and Apple Maps URL schemes to launch the maps application with directions to the coordinates we supply. If we are running on iOS then we launch Apple Maps using the `maps://` scheme, and if we are running on Android then we launch Google Maps using the `geo:` scheme. Now when the user triggers this function, a map should pop up with directions back to their marked campsite.

That's it! We've finished our maps functionality. A user should now be able to view the map, set their location, and trigger directions back to their campsite. We still need to take care of one more important thing and that's saving the data so that it is available when the user comes back to the application. We will be handling that, as well as saving the data from the forms, in the next lesson.

NOTE: If you are using Geolocation in an application that you are submitting to App Stores, you should include usage descriptions as described in the Geolocation documentation.

Lesson 6: Saving and Retrieving Data

If you've been through some of the other applications, then you will be pretty familiar with creating a Data service. This one is a little different though because in the other applications we were only ever storing one set of data, but in this application, we are going to have to save:

- Camp Coordinates
- Camp Details
- My Details

The idea is still more or less the same, but it's going to look a little different. We *could* just store all of the data in one big object under one key as we have been with previous applications, but since we are dealing with different types of data I think it makes more sense to separate them out. We already have our forms and our maps page sending data to our data service so we just need to handle saving it, and we also need to make a few modifications to load that data back in again. Let's start out by implementing the data service.

> Modify `src/app/services/data.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { Storage } from "@ionic/storage";

import { Location } from "../interfaces/location";
```

```
import { CampDetails } from "../interfaces/camp-details";
import { MyDetails } from "../interfaces/my-details";

@Injectable({
  providedIn: "root",
})
export class DataService {
  constructor(private storage: Storage) {}

  setMyDetails(data: MyDetails): void {
    this.storage.set("mydetails", data);
  }

  setCampDetails(data: CampDetails): void {
    this.storage.set("campdetails", data);
  }

  setLocation(data: Location): void {
    this.storage.set("location", data);
  }

  getMyDetails(): Promise<MyDetails> {
    return this.storage.get("mydetails");
  }

  getCampDetails(): Promise<CampDetails> {
    return this.storage.get("campdetails");
  }
```

```
  getLocation(): Promise<Location> {
    return this.storage.get("location");
  }
}
```

Storage is Ionic's generic storage service, and it handles storing data in the best way possible whilst providing a consistent API for us to use.

When running on a device, and if the SQLite plugin is available (which we installed earlier), it will store data using a native SQLite database. Since the SQLite database will only be available when running natively on a device, `Storage` will also use `IndexedDB`, `WebSQL`, or standard browser `localStorage` if the SQLite database is not available.

It's best to use SQLite where possible because the browser-based local storage is not completely reliable and can potentially be wiped by the operating system. Having your data wiped randomly is obviously not ideal.

In other applications, we would usually just have two functions, one for getting data and one for saving data. In this application, though we are setting three different sets of data, so we create three **set** functions and three **get** functions. The set functions take in the data we pass it and store it (after converting it to a JSON string), and then the get functions will retrieve those values from the database. The get functions will return a promise which resolves with the data, rather than the data directly, so we will need to set up a handler for that when we use it.

Now that we have our data service set up, we just need to load in the data that is saved when we reopen the application. So now we are going to make changes to the Location,

My Details and Camp Details pages as they all have some data they need to be loaded in.

We'll start off with the location page.

> Modify the `ngOnInit` method in `src/app/location/location.page.ts` to reflect the following:

```
  async ngOnInit() {
    const location = await this.dataService.getLocation();

    this.map.getLoadedStatus().subscribe((status) => {
      if (status === "loaded" && location) {
        this.location = location;
        this.map.changeMarker(location.latitude,
location.longitude);
      }
    });
  }
```

First, we grab the users location from storage - if there is a location available we will use it in a moment. We want to use that location to add a marker on the map, but we can't just do it immediately because we need to wait for Google Maps to finish loading. To achieve this, we subscribe to the loaded status that our Google Maps component provides (notice that we can call our `public` methods inside of our component just by having a reference to the component). When we see the `loaded` status we set the marker on the map.

Next, let's take care of loading in the form data for our Camp Details page.

> Modify the `ngOnInit` method in `src/app/camp-details/camp-details.page.ts` to reflect the following:

```
  async ngOnInit() {
    const campDetails = await this.dataService.getCampDetails();
    const formControls = this.campDetailsForm.controls;

    if (campDetails) {

formControls.gateAccessCode.setValue(campDetails.gateAccessCode);

formControls.ammenitiesCode.setValue(campDetails.ammenitiesCode);

formControls.wifiPassword.setValue(campDetails.wifiPassword);
      formControls.phoneNumber.setValue(campDetails.phoneNumber);
      formControls.departure.setValue(campDetails.departure);
      formControls.notes.setValue(campDetails.notes);
    }

    this.campDetailsForm.valueChanges.pipe(
      debounceTime(1000)
    ).subscribe(() => {
      this.dataService.setCampDetails(this.campDetailsForm.value);
    });

  }
```

You can see we've added in another call to retrieve some data from the data service here and then we're doing some stuff with it. Remember before how I said you can supply an initial value when creating your group with `FormBuilder`, i.e:

```
gateAccessCode: ['value here']
```

You might expect that you could just load in the data and then build your form using the saved values as the initial values. Unfortunately, it doesn't quite work like this. The data being retrieved from memory is asynchronous, so although the data is retrieved very quickly there is some wait time involved. The `Form Builder` group needs to be created instantly, otherwise, you will receive errors. Instead we create the `FormBuilder` group right away (in the `constructor`), and then we grab a reference to the forms controls using `this.campDetailsForm.controls`. We can then access the `setValue` method on each of the controls we created to set the value to the saved value.

We also commented out the call to the data service in the `ngOnInit` method previously, which we have rectified now. Now we just need to do the same for the My Details page.

> Modify the `ngOnInit` method in `src/app/my-details/my-details.page.ts` to reflect the following:

```
async ngOnInit() {

  const myDetails = await this.dataService.getMyDetails();
  const formControls = this.myDetailsForm.controls;
```

```
    if(myDetails){

formControls.carRegistration.setValue(myDetails.carRegistration);

formControls.trailerRegistration.setValue(myDetails.trailerRegistra


formControls.trailerDimensions.setValue(myDetails.trailerDimensions

      formControls.phoneNumber.setValue(myDetails.phoneNumber);
      formControls.notes.setValue(myDetails.notes);
    }

    this.myDetailsForm.valueChanges.pipe(
      debounceTime(1000)
    ).subscribe(() => {
      this.dataService.setMyDetails(this.myDetailsForm.value);
    });

  }
```

We've uncommented the calls to save the data in the data service in both the Camp Details and My Details classes above, but we also need to uncomment it in the Location class as well.

> Uncomment the `dataService` call in the `setLocation()` method in `src/app/location/location.page.ts`:

```
  async setLocation(): Promise<void> {
    const overlay = await this.loadingCtrl.create({
      message: "Setting current location...",
    });

    overlay.present();

    try {
      const newLocation = await Geolocation.getCurrentPosition({
        enableHighAccuracy: true,
        timeout: 10000,
      });

      this.location = {
        latitude: newLocation.coords.latitude,
        longitude: newLocation.coords.longitude,
      };

      this.map.changeMarker(this.location.latitude,
this.location.longitude);
      this.dataService.setLocation(this.location);

      const alert = await this.alertCtrl.create({
        header: "Location set!",
        message:
          "You can now find your way back to your camp site from
anywhere by clicking the button in the top right corner.",
        buttons: [
```

```
        {
          text: "Ok",
        },
      ],
    });

    overlay.dismiss();
    alert.present();
  } catch (err) {
    console.log(err);
    overlay.dismiss();
  }
}
```

and that's it! Any data that is entered in the application should remain there when the application is reloaded now.

We're getting pretty close to finishing this application, we obviously still need to add some styling (although it looks pretty groovy already). We will finish the rest of the application in the next lesson.

Lesson 7: Styling

This application is pretty stock standard, so there's not going to be too much customisation going on here and this lesson is going to be super quick. But we do want to add a few little final touches.

Let's start by modifying our primary colour value.

> Modify the `--ion-color-primary` variable in `theme/variables.scss` to reflect the following:

```
--ion-color-primary: #54a0ff;
```

Ionic also comes with styling for a **dark mode** by default, which will be enabled if the users preferences are set to use dark mode where available. We will be removing that, as it will mess with the styles that we will set up.

> Remove everything inside of `@media (prefers-color-scheme: dark) { }` in `src/theme/variables.scss`:

Alternatively, you could supply your own different set of styles that you want to be activated when dark mode is switched on. I have this as a homework exercise in the next lesson, so if you would like to have a go at implementing your own dark mode theme you should keep the current contents of `@media (prefers-color-scheme: dark) { }`

so that you have a base to work from.

We also just want to apply some general styles to our whole application.

> Add the following styles to `global.scss`:

```
ion-title img {
  max-height: 35px;
  margin-top: 4px;
}

ion-content {
  --ion-background-color: var(--ion-color-primary);
}

ion-label {
  font-weight: bold;
}

ion-input,
ion-textarea,
ion-datetime {
  margin-top: 5px;
  margin-bottom: 20px;
  background-color: #fff !important;
  border: 1px solid #cecece;
  --padding-start: 15px !important;
```

```
  --padding-end: 15px !important;
  --padding-top: 15px !important;
  --padding-bottom: 15px !important;
}
```

The main thing we are trying to achieve here is to add a bit of styling to our input fields. We have also used a different background colour for the `ion-content` area, so now the white used for our inputs will stand out clearly.

We are also going to add a bit of a shadow to our tab bar and remove it's border which will make things look a little snazzier.

> Add the following styles to `src/app/home/home.page.scss`:

```
ion-tab-bar {
  border: none;
  box-shadow: 0px 8px 20px 0px rgba(25, 25, 47, 0.5);
}
```

The application should now look like this:

Find your camp
Map
Satellite
Brooklyn Av
Rothesay Av
A15
N Esplanade
Patawalonga River
Old Gu
Holdfast Bay
Glenelg N
Tod St
Sturt St
Sammy's on the Marina
The Watermark
Wigley Reserve
Oaks Glenelg Plaza Pier Suites
Sussex St
Nile St
Coles Glenelg
Glenelg Beach
Jetty Rd
Google
Map data ©2020
Terms of Use
Location
My Details
Camp Details

CamperMATE
Car Registration
IONIC
Trailer Registration
ISCOOL
Trailer Dimensions
8 x 16ft
Phone Number
5555555
Notes
Location
My Details
Camp Details

This is the single smallest lesson in the entirety of this book, but, we're done! The CamperMate application is now completely finished.

Conclusion

Congratulations on making it through the Camper Mate tutorial. We've learned a lot through developing this application, but the main takeaways are:

- Creating forms and capturing user data
- Implementing Google Maps and creating a custom component to handle that
- Handling network connectivity
- Creating a tabs layout
- Using nested routes with `<ion-router-outlet>`
- Saving and retrieving multiple sets of data

There's always room to take things further though, especially when you're trying to learn something. Following tutorials is great, but it's even better when you figure something out for yourself. Hopefully, you have enough background knowledge now to start trying to extend the functionality of the application by yourself, here are a few ideas to try out:

- Use Validators on the Phone Number and Date input fields to ensure valid data is input **[MEDIUM]**
- Instead of allowing one marker at a time, keep markers on the map for every place the user has checked in at (and only navigating to the most recently added marker) **[HARD]**
- Add a field to 'Camp Details' that allows the user to take a photo and display it there (see Snapaday application for help) **[HARD]**

Remember, the Ionic documentation is your best friend when trying to figure things out.

What next?

You have a completed application now, but that's not the end of the story. You also need to get it running on a real device and submitted to app stores, which is no easy task. The final sections in this book will walk through how to take what you have done here, and get it onto the app stores so make sure to give that a read.

Camper Chat

Lesson 1: Camper Chat Introduction

This one is the big one, it's the last application we will be building and I think it's a good one to finish on. As with all the other applications in this book, there is no need to complete the other applications before doing this one as everything will still be explained, even if the same thing has been explained in other applications, but the complexity is turned up a bit in this one and I won't be spending as long on the basics. So, if you're not that comfortable with Ionic yet you might find one of the other applications easier to start on.

In this section, we will be building **Camper Chat**, which will essentially be a live chat application. Users will be able to log in with their Facebook account and chat with anybody else who is using the application. We will be using **Firebase** for the backend, which will allow us to store our data remotely and provide live updates whenever new data is added to the database. Firebase will also help us facilitate authentication for our users, and it will allow us to automatically log users in when they return to the application if they have logged in previously.

To give you a more precise definition, the exact features of the application will be:

- Users can log in with a Facebook account to use the application
- Users can leave messages that all other users can see and respond to (in real-time)
- The users Facebook display picture and name will be used in the application

- Users can log out
- Users can view an 'About' page.

Given these requirements, we will be learning about a bunch of different things including:

- Navigation and restricting access to routes
- Using a sliding menu
- Using Firebase/Firestore to store data
- Using Firebase for authentication
- Simple animations
- Using the Facebook API for native and web authentication
- Configuring native projects for plugin integrations
- Updating and displaying data in real-time

and here are some screenshots to help you get an idea of what the application will look like:

CamperCHAT
Camper Chat allows you to talk to other caravaners, RV'ers, road trippers and campers near you. Use Camper Chat to give and receive tips, ask for help with your flat tyre, or just have a friendly chat.
Connect with Facebook

CamperCHAT
Joshua Morony
We're staying for a few days or so
Kathy Sara
We just left there! The parks were all pretty full
Kathy Sara
but there were some good free spots you could check out 😄
Joshua Morony
Thanks!
type message...

Please do keep in mind that this will still be a very basic implementation of a chat application, and it is intended to introduce you to some key concepts - not to build a complete/robust solution for a chat application (e.g. Facebook's "Messenger" or "WhatsApp"). You could certainly extend this application to include more advanced features like individual chat rooms if you invest more time into learning about how to use Firebase at a more advanced level.

Lesson Structure

1. Getting Ready
2. Login Page and Sliding Menu Layout
3. Using Facebook for Authentication
4. Creating and Displaying Messages & Navigation
5. Using Firestore to Store Messages
6. Firestore Security Rules
7. Styling & Animations

Ready?

Now that you know what you're in for, let's get to building it!

Lesson 2: Camper Chat Getting Ready

In this lesson, we are going to prepare our application for the journey ahead. We are going to generate the application, and we are also going to set up all of the components, plugins, and routes that we need. The idea is to get all of the general scaffolding required for most projects out of the way so we don't have to mess around with creating files and configuring things throughout the rest of the lessons.

At the end of this first lesson, we should have a nice skeleton application set up with everything we need to start diving into coding.

A good rule of thumb before starting any new application is to make sure you have the latest version of the Ionic CLI installed, so if you haven't done it recently then make sure to run:

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

If your environment is not set up correctly, you may run into permission errors when installing this package globally. If you are having permission issues, you can use the Ionic teams guide to solving these issues: Resolving permission errors.

If you do not already have a recent version of Node/npm set up on your machine, make sure to complete the Environment Setup guide first.

Generate a new application

We will be using the blank starter template for this application which, as the name implies, is basically an empty Ionic project. It comes with one page built-in called **HomePage** but we will be adding an additional page.

> Run the following command to generate a new application

```
ionic start chat blank --type angular
```

When you are prompted with the following:

```
? Integrate your new app with Capacitor to target native iOS and
Android? (y/N)
```

Say **Yes**, as we will be utilising Capacitor for native builds. If you do not set up Capacitor when first creating your application, you can always do it manually later as well by running the following command:

```
ionic integrations enable capacitor
```

NOTE: You should still install Capacitor in this project even if you do not intend to build for iOS or Android, as we will still be using Capacitor for the web platform as well.

> Make the new project your current working directory by running the following command:

```
cd chat
```

Your project should now be generated - now you can open up the project folder in your favourite editor. You can take a look at how your application looks by running the following command:

```
ionic serve
```

which for now should look something like this:

Blank

Ready to create an app?

Start with Ionic UI Components

Create the Required Components

This application is going to have a few different pages, we are going to be using a login page, the main home page and an about page. The home page has already been generated for us, so we will just need to create the login page and the about page.

NOTE: You will need to stop serving your application in order to run these commands. If your application is currently being served through the command line, make sure to hit `Ctrl + C` first to stop the process.

> Run the following command to generate the Login page:

```
ionic g page Login
```

> Run the following command to generate the About page:

```
ionic g page About
```

Create the Required Services

As well as our pages, we are also going to create an `Auth` service which will handle everything related to authentication for us, and a `Messages` service which will handle storing and retrieving our message data, as well as storing a few values that we will grab from the Facebook API. We are also going to create an `AuthGuard` service that will help us protect particular routes from access by the user if they are not logged-in.

> Run the following command to generate the Auth service:

```
ionic g service services/Auth
```

NOTE: Notice that we use `services/Auth` instead of just `Auth` this time. This is because we want to generate all of our services inside of a folder called `services`.

> Run the following command to generate the Messages service:

```
ionic g service services/Messages
```

> Run the following command to generate an AuthGuard service:

```
ionic g service services/AuthGuard
```

Configure the Routes

As we discussed in the basics section, we need to define routes so that our application can match up the current URL to a particular component that you want to display. Generating the application and using the generate commands do most of the work for us, as routes are automatically injected into the **app-routing.module.ts** file. However, we will need to make some slight modifications to these.

> Modify `src/app/app-routing.module.ts` to reflect the following:

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

```
import { PreloadAllModules, RouterModule, Routes } from
"@angular/router";

const routes: Routes = [
  {
    path: "",
    redirectTo: "home",
    pathMatch: "full",
  },
  {
    path: "login",
    loadChildren: () => import("./login/login.module").then((m) =>
m.LoginPageModule),
  },
  {
    path: "home",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
  },
  {
    path: "about",
    loadChildren: () => import("./about/about.module").then((m) =>
m.AboutPageModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
```

```
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Create the Interfaces

We will eventually be using a custom `interface` to represent a `Message` and `User` data type. Unfortunately, there is no generate command for this, so you will need to create it manually.

> Create a new folder inside of the app folder called `interfaces`

> Create a new file inside of the interfaces folder called `message.ts` and add the following:

```
export interface Message {
  uid: string;
  displayName: string;
  displayPicture: string;
  message: string;
  created: string;
}
```

This will represent the messages we want to send to our Firestore database, and receive from our Firestore database. We will discuss this in more depth later, but the data in the type above allows us to have a user id associated with the message, a name, their profile picture, the message they sent, and the time that it was created.

> Create a new file inside of the interfaces folder called `user.ts` and add the following:

```
export interface User {
  uid: string;
  displayName: string;
  displayPicture: string;
}
```

This interface represents the information we want to store about a logged-in user, which for our purposes is just their user id, their display name, and their profile picture.

Install Firebase

In order to integrate with Firebase, we will be making use of the Firebase Javascript SDK. As you will see later, when going through the install steps for Firebase you will be told to use a `<script>` tag, but we will be installing an `npm` package instead. This will integrate more nicely into our Angular environment:

```
npm install firebase
```

We will handle setting up everything else for Firebase later.

Configure Capacitor

At this point, you should already have Capacitor added to your project. However, if you did not respond **yes** to the prompt about installing Capacitor when initially creating the project, you should manually set it up now.

> Run the following command (if required) to enable Capacitor:

```
ionic integrations enable capacitor
```

Keep in mind that by default your application will be given a **Bundle ID** of `io.ionic.starter`. This Bundle ID is used to identify your native iOS/Android builds, and you will eventually want to change this to something unique to you or your company. You can change it later if you wish, but it is easier to do before we add the iOS and Android platforms. You may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`. If you don't intend on eventually modifying and publishing this project, then this step doesn't matter.

If you look at the **capacitor.config.json** file, you will notice that the `webDir` is listed as `www`. This is the folder that Capacitor will look to when it is copying over the code for your application into the native projects. It doesn't care about the rest of your source code, it only cares about the built output (i.e. the code that is actually run through the browser). By default, the `www` folder will not exist in your project until you perform a build of your

application manually. We are going to do that now.

> Run the following command to generate a build of your application:

```
ionic build
```

It is good to use the standard `ionic build` command whenever you want to test your application on a device during development, but keep in mind that this creates a development build. When you are building the final version of your application, or if you want to test a production version of your application, you should run:

```
ionic build --prod
```

which will create an optimised production build.

Add Native Platforms

By default, the iOS and Android platforms will not be enabled in your project. If you want to add these platforms you can do so now (or you can do it later if you prefer). Remember, building for iOS will require Xcode and building for Android will require Android Studio - if you want to add these platforms now, you should make sure you have followed the instructions in the Generating an Ionic Application lesson to install Xcode, Android Studio, and their dependencies.

```
ionic cap add ios
```

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your application code and dependencies over to the native projects at any time by running:

```
ionic cap sync
```

When you make changes to your application, you will need to run this command to copy the changes over to Capacitor (otherwise your changes won't appear when you run the application on iOS or Android).

Set up Root Component

The default starter templates for Ionic utilise **Ionic Native** to handle hiding the **Splash Screen** and styling the **Status Bar**. Whilst you can still use **Ionic Native** and **Cordova** plugins inside of a Capacitor project, Capacitor provides this functionality through default APIs, so we are going to use those instead. It is also important to remember that the Cordova Splash Screen plugin is not compatible with Capacitor, so make sure that you do not install it in your project.

> **Modify `src/app/app.component.ts` to reflect the following:**

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor() {
    this.initializeApp();
  }

  async initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#54a0ff" }).catch((err)
=> {
      console.warn(err);
    });
  }
}
```

You can also use the `StatusBar` API to hide the status bar entirely, but in this case we are styling it with a colour that will match our application. The status bar is the bar at the top of native applications that display the time, battery level, etc.

Set up Ionic Storage

We will be making use of the Ionic Storage API in this application. Ionic provides a simple key/value storage API that we can use in our applications to store data - it provides a consistent API and will automatically use the best storage mechanism available. That means that if we install the SQLite plugin in our project, it will store our data in native storage (rather than browser local storage which is more likely to get wiped by the operating system randomly). If native storage is not available, it will default back to browser storage.

To enable Ionic storage, you need to install it.

> Install Ionic Storage with the following command:

```
npm install @ionic/storage
```

and you will also need to add it to your root module file.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass:
IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Notice that we have imported `IonicStorageModule` and we have also added it to the `imports` array. We have also removed the `SplashScreen` and `StatusBar` providers from Ionic Native since we are not using them anymore.

Set up Plugins

We are going to set up any native plugins that we need now. Remember that when installing Cordova plugins in a Capacitor project you should install them using the `npm install` command, **not** with `ionic cordova plugin add`.

> Run the following command to add the SQLite plugin:

```
npm install cordova-sqlite-storage
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

> Run the following commands to add the Facebook plugin:

```
npm install cordova-plugin-facebook4
```

```
npm install @ionic-native/facebook
```

Since we are using Ionic Native, we will also need to add this plugin into the **app.module.ts** file as a provider.

> Modify `src/app/app.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { RouteReuseStrategy } from "@angular/router";

import { IonicModule, IonicRouteStrategy } from "@ionic/angular";
```

```
import { IonicStorageModule } from "@ionic/storage";

import { AppComponent } from "./app.component";
import { AppRoutingModule } from "./app-routing.module";
import { AuthGuardService } from "./services/auth-guard.service";

import { Facebook } from "@ionic-native/facebook/ngx";

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
AppRoutingModule, IonicStorageModule.forRoot()],
  providers: [
    Facebook,
    AuthGuardService,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy },
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

We have also taken this opportunity to add the `AuthGuardService` as a provider.

Remember, after making changes to the native plugins in your project you will need to run:

```
ionic cap sync
```

This will copy the new plugins you have installed over to the native projects.

Set up Images

When building this application we are going to be making use of a few images. I've included these in your download pack but you will need to set them up in the application you generate.

> Copy the images folder in the download pack for this application from `src/assets` to your own `src/assets` folder

Summary

That's it! We're all set up and ready to go, now we can start working on the interesting stuff. Make sure to test your application by running `ionic serve` every now and then, it's easier to catch and fix errors along the way than waiting until later. If you ever get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again.

Lesson 3: Login Page and Sliding Menu Layout

We're going to be implementing a really common page flow in this application, and that's a login page that leads to the main application. In pretty much any application that requires users to be authenticated, there is going to be some authentication screen shown first, and only once the user successfully logs in will they be able to access the main application.

In our case, we are going to have a login page that will have a "Login with Facebook" button, which will then lead to a page with a sliding side menu layout. Our login page is pretty simple (for now at least) so let's get started by building that.

> Modify `src/app/login/login.page.html` to reflect the following:

```
<ion-content>
  <div class="container">
    <img src="./assets/images/logo.png" class="logo" />

    <div class="login-description">
      <p>
        Camper Chat allows you to talk to other
        <strong>caravaners, RV'ers, road trippers and
campers</strong> near you. Use Camper Chat to
        give and receive tips, ask for help with your flat tyre,
or just have a friendly chat.
      </p>
    </div>
```

```
    <a (click)="login()"
      ><img
        src="./assets/images/facebook-button.png"
        alt="Login with Facebook"
        class="login-button"
    /></a>
  </div>
</ion-content>
```

This defines the template for our login page, and if this isn't the first application you've built with Ionic (and hopefully it isn't) then you might notice we've left the usual `<ion-header>` and `<ion-toolbar>` out of the template. We have no need to display a title and we don't need to display a back button for navigation or anything like that, so it's perfectly fine to just leave it out.

We've also attached a `(click)` handler to a Facebook login button image. Eventually we will have this `login` function trigger the authentication with Facebook, but for now, we are just going to have it trigger a page change to the main page so that we can finish setting up our layout. Let's set up the class for this component now so that we can set up this page transition.

> Modify `src/app/login/login.page.ts` to reflect the following:

```
import { Component, OnInit } from "@angular/core";
import { MenuController, LoadingController, NavController } from
```

```
"@ionic/angular";
import { AuthService } from "../services/auth.service";
import { debounceTime } from "rxjs/operators";

@Component({
  selector: "app-login",
  templateUrl: "./login.page.html",
  styleUrls: ["./login.page.scss"],
})
export class LoginPage implements OnInit {
  private loading: HTMLIonLoadingElement;

  constructor(
    private menu: MenuController,
    private navCtrl: NavController,
    private loadingCtrl: LoadingController,
    private authService: AuthService
  ) {}

  async ngOnInit() {}

  ionViewDidEnter() {
    this.menu.enable(false);
  }

  login() {
    this.menu.enable(true);
    this.navCtrl.navigateRoot("/home");
  }
```

```
  async showLoading() {}
}
```

As you can see in the code above, we've injected and set up our `NavController` and we're using it to navigate to `/home` when the login function is called.

The `MenuController` is used to programmatically interact with the sliding menu we will be adding to the application. We will discuss that in more detail in a moment, but we don't want the menu to be used on the login page so we use the `enable` function on the menu to disable it for this page. When we switch to our Home Page we enable it again. Since we used the `ionViewDidEnter` hook, the menu will be disabled every time the login page is activated.

Since the user will be able to log in, we also want them to be able to log out. The log out function will be made available through the side menu, and the side menu will be added to our root component's template. Let's set up a little scaffolding for the root component now.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { MenuController } from "@ionic/angular";
import { AuthService } from "./services/auth.service";
import { MessagesService } from "./services/messages.service";
```

```
const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(
    private authService: AuthService,
    private messagesService: MessagesService,
    private menu: MenuController
  ) {
    this.initializeApp();
  }

  initializeApp() {
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#54a0ff" }).catch((err)
=> {
      console.warn(err);
    });
  }

  async logout() {}
```

```
}
```

We're not actually doing much here yet - the logout function is currently empty - but we are setting up the basic services we are going to be relying on. We're importing and setting up references to our `Auth` service, which we will use for logging in and out, the `Messages` service, and the `MenuController`, which will allow us to interact with the sliding menu we are creating.

I mentioned that we would set the side menu up in the template for the root component, let's do that now.

> Modify `src/app/app.component.html` to reflect the following:

```
<ion-app>
  <ion-menu contentId="main-content" side="start">
    <ion-header class="ion-no-border">
      <ion-toolbar color="primary">
        <ion-title>Menu</ion-title>
      </ion-toolbar>
    </ion-header>
    <ion-content>
      <ion-list lines="none">
        <ion-menu-toggle auto-hide="false">
          <ion-item button routerLink="/home"
routerDirection="root">
```

```
            <ion-icon slot="start" name="chatbubbles"></ion-icon>
            <ion-label> Chat </ion-label>
          </ion-item>
          <ion-item button routerLink="/about"
routerDirection="root">
            <ion-icon slot="start" name="information-circle">
</ion-icon>
            <ion-label> About </ion-label>
          </ion-item>
          <ion-item button (click)="logout()">
            <ion-icon slot="start" name="power"></ion-icon>
            <ion-label> Logout </ion-label>
          </ion-item>
        </ion-menu-toggle>
      </ion-list>
    </ion-content>
  </ion-menu>

  <ion-router-outlet id="main-content"></ion-router-outlet>
</ion-app>
```

There's some really interesting stuff going on here. This part of the code above will likely look pretty familiar to you:

```
<ion-router-outlet id="main-content"></ion-router-outlet>
```

Except for one bit, and that's that we have added `id="main-content"` to the `<ion-router-outlet>` (as well as a matching `contentId` to the `<ion-menu>`). In order to behave correctly, the menu needs to know what content to attach itself to. What we're doing here is essentially telling the menu where our content is, which in this case is our entire application. Everything else inside of `<ion-menu>` is just the content that the menu will contain, and all we're doing is creating a list of buttons that will serve as navigation options for the user to click on. Each of these options will link to a specific route in the application, and we use the `ion-menu-toggle` component to automatically toggle the menu to close whenever an option is selected.

The final piece of this layout puzzle is to create the home page, let's start off by defining the template.

> Modify `src/app/home/home.page.html` to reflect the following:

```
<ion-header class="ion-no-border">
  <ion-toolbar color="primary">
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title><img src="assets/images/logo.png" /></ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <ion-list lines="none">
    <ion-item>
```

```
      <ion-avatar slot="start">
        <img src="" />
      </ion-avatar>
      <div class="chat-message">
        <ion-note>Name here</ion-note>
        <p>Message here</p>
      </div>
    </ion-item>
  </ion-list>
</ion-content>

<ion-footer>
  <ion-toolbar color="primary">
    <ion-textarea
      (keyup)="handleKeyup($event)"
      [(ngModel)]="chatMessage"
      class="chat-input"
      placeholder="type message..."
    >
    </ion-textarea>

    <ion-buttons slot="primary">
      <ion-button (click)="sendMessage()"
[disabled]="addingMessage">
        <ion-icon [hidden]="addingMessage" slot="icon-only"
name="send"></ion-icon>
        <ion-spinner [hidden]="!addingMessage" name="crescent">
</ion-spinner>
```

```
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-footer>
```

We have our `<ion-header>` added back in this template and we've added the `<ion-menu-button>` component which will add a button to allow the user to toggle the menu on and off (the menu can also be opened and closed by swiping, which is why we needed to disable it on the login page even though we didn't add a menu button).

Inside of the content area we create a list with just a single `<ion-item>` for now (later we will use `*ngFor` to display all of the available messages), and we're also using `<ion-avatar>` which allows us to attach a nicely styled picture to the item. Eventually, we will also add the name of the user who posted the message, and their message here as well.

We're also using an `<ion-toolbar>` in the footer. This allows us to have an area for our message input which will stay stuck at the bottom of the screen even as the list scrolls. Inside of the toolbar we've added an input field which has two-way data binding set up on `chatMessage` by using `[(ngModel)]`. This means that when this value changes, the value for `this.chatMessage` in **home.page.ts** (which we are about to create) will also be updated (and vice versa). Then we just add a button which will eventually send the message that is entered. Also notice that we bind the `keyup` event for the `<ion-textarea>`. We are going to allow users to submit the message by hitting the "enter" key as well as by clicking the send button - in the next step, we will implement the `handleKeyup` method to check if the "enter" key was pushed.

Now let's create the class definition for our **home.page.ts** file.

> Modify `src/app/home/home.page.ts` to reflect the following:

```
import { Component, ViewChild, ElementRef, OnInit } from
"@angular/core";
import { IonContent, IonList, NavController } from
"@ionic/angular";
import { Subscription } from "rxjs";
import { Message } from "../interfaces/message";
import { User } from "../interfaces/user";
import { AuthService } from "../services/auth.service";
import { MessagesService } from "../services/messages.service";

@Component({
  selector: "app-page-home",
  templateUrl: "./home.page.html",
  styleUrls: ["./home.page.scss"],
})
export class HomePage implements OnInit {
  @ViewChild(IonContent, { static: false }) contentArea:
IonContent;
  @ViewChild(IonList, { static: true, read: ElementRef })
chatList: ElementRef;

  public user: User = { uid: "", displayName: "", displayPicture:
"" };
```

```
  public chatMessage: string = "";
  public messages: Message[] = [];
  public addingMessage: boolean = false;
  private mutationObserver: MutationObserver;

  private loggedInSubscription: Subscription;
  private messagesSubscription: Subscription;
  private detachListener: Function;

  constructor(
    public messagesService: MessagesService,
    public authService: AuthService,
    private navCtrl: NavController
  ) {}

  ngOnInit() {}

  ngOnDestroy() {}

  handleKeyup(ev: KeyboardEvent) {
    if (ev.key === "Enter") {
      this.sendMessage();
    }
  }

  async sendMessage() {}
}
```

We're importing and injecting our `Messages` and `Auth` services again and we have also set up a few class member variables. We set up the `this.chatMessage` class member that our input field from the template will be bound to. We are also using `@ViewChild` to grab references to elements contained in our template. By passing `IonContent` (which is the class for the `<ion-content>` component) to `@ViewChild` we are able to get a reference to our content area. Similarly, by passing `IonList` (which is the class for the `<ion-list>` component) we are able to get a reference to the `<ion-list>` in our template. For the list, we actually want a reference to the DOM element itself, so we supply the `read` option and tell it to return as an `ElementRef`. We will be using these references in conjunction with the `mutationObserver` later, but we don't need to worry about that just yet.

We have a `user` member variable that we will eventually set using data we obtain from the `Auth` service about the user that is currently logged in, but for now we just set it to an empty user object.

We have also set up a few member variables to keep track of the observables and listener functions that we will be making use of. The reason we do this is so that in the `ngOnDestroy` method (which is triggered when the component is destroyed) we can unsubscribe from these observables and remove any references to stuff we no longer need. This keeps things tidy, and stops weird issues due to having multiple subscriptions to the same thing running (e.g. if you destroyed the component but then came back to it again later).

The one method that we have fully implemented here already is the `handleKeyup` method. All we do is check to see if it was the `Enter` key that was pressed, and if it is we call the `sendMessage` method.

If you run your application using:

```
ionic serve
```

you should have two pages now that look like this:

CamperCHAT
Name here
Message here
type message...

Camper Chat allows you to talk to other **caravaners, RV'ers, road trippers and campers** near you. Use Camper Chat to give and receive tips, ask for help with your flat tyre, or just have a friendly chat.

Connect with Facebook

As is usually the case, it doesn't look fantastic right now but the basic structure for the layout we need is there. We have a login page with a login button, which leads to the main page with a sliding menu.

In the next lesson, we're going to start integrating the **Facebook API** so that we can have users logging in for real (which will also provide us some data to use in the application like the user's name and profile picture).

Lesson 4: Facebook Authentication with Firebase

In this lesson, we'll be integrating the Facebook plugin from Ionic Native which will allow us to authenticate users in our application, and it will also allow us to do a few other things like grabbing the user's profile information. There's a ton more you can do with the Facebook API that I won't be covering though, so make sure to check out the documentation.

We will only be using the Facebook plugin to allow us to create a native authentication flow, we are actually using it in conjunction with the Facebook sign-in method that Firebase provides. We can authenticate with Facebook easily through the Firebase Javascript SDK, but this is only for a web-based authentication flow. This means that a browser will be used to sign the user in, but if you have your application deployed to iOS or Android, you probably want to use the native Facebook application for authentication so that the user does not need to enter in their email/password manually.

To allow the user to use a web-based authentication flow when running the application on the web (e.g. as a PWA), and a native authentication flow when deployed as a native application, we will automatically detect if the application is running natively and if it is we will utilise the Facebook plugin to get the authentication information for the user and pass it along to Firebase. If the application is not running natively, then we will just use the normal web-based Firebase/Facebook sign-in flow.

Using social authentication like the Facebook API has a lot of advantages. It saves us from having to create our own backend with an authentication system (which is something that should not be undertaken lightly), it saves the user from having to create another new

account to use the application (they can just log in with one tap), and it provides a bunch of useful data and integrations. This isn't to say that social login is always the best authentication method, but it certainly has its advantages.

Let's walk through how to set up the Facebook plugin and then we can look at integrating into our application.

IMPORTANT: In order to test the native authentication flow, you will need to run the application on an iOS or Android device (or simulator). If you are unsure of how to run an application on your device, make sure to read the **Building & Submitting** section at the end of this book. There are some extra steps you need to take in order to get the application running on a device successfully.

Setting up a Facebook App

With a lot of plugins, all you need to do is install the plugin and you are good to go. This plugin is a little trickier. We are going to need to supply the plugin with an **App Name** and an **App ID** from Facebook to configure it (as well as a few other values). To do that, we need to create an application on Facebook's developer platform.

First, you will need to go to developers.facebook.com and sign up if you haven't already. The interfaces for these developer portals often change, so there might be some slight discrepancies in what I am describing and what you are seeing. Even though the interface might change a bit, the general steps should remain the same:

1. Go to developers.facebook.com and create an account if necessary
2. Go to **My Apps** and click **Create App**
3. Choose **Build Connected Experiences**

4. Add a **Display Name** and click **Create App**
5. Under **Add Products to Your App** click **Set Up** on **Facebook Login**
6. Select **Settings** from the left side menu (the main settings **immediately under "Dashboard"**, **not** the settings under "Facebook Login" - ignore the "Quickstart" options we are presented with initially)
7. Select **Basic** from the settings dropdown
8. From here, you will be able to see your **App ID** and **App Secret** - make a note of both of these for later (or you can just come back to this page later)
9. Scroll to the bottom and click **+ Add Platform**
10. Select **iOS** (if you are building for iOS) and add the **Bundle ID** of your application (e.g. `com.yourcompany.yourapp` or `io.ionic.starter`)
11. Select **+ Add Platform** again and then select **Android** (if you are building for Android) and add your **Bundle ID** to the **Google Play Package Name**
12. Click **Save Changes**

If you are building for **Android** you will also need to create a **key hash** of your **keystore** file that will be used to sign your Android application (we talk more about keystore files in the **Building & Submitting** section). You will need to provide a key hash for the keystore file that you use for the production version of your application, but throughout development, you can just provide a key hash of the default `debug.keystore` file that is used by **Android Studio** to sign development versions of your application.

If you are using a **Mac**, you can use the following command:

```
keytool -exportcert -alias androiddebugkey -keystore
~/.android/debug.keystore | openssl sha1 -binary | openssl base64
```

or if you are using **Windows** you can use the following command (this will require OpenSSL):

```
keytool -exportcert -alias androiddebugkey -keystore
%HOMEPATH%\.android\debug.keystore | openssl sha1 -binary |
openssl
base64
```

You will need to enter the password: `android`. Once you do this, your key hash will be displayed on the screen and you can add it to the **Key Hashes** field in the Facebook developer portal (under the **Android** platform that we just added). Remember that you will need to update this key hash later (or add an additional one) to reflect the keystore file that is used to sign the production version of your application.

NOTE: If you do not already have a JDK installed you may be asked to do so when performing this step. Typically, this isn't required because Android Studio comes bundled with its own JDK. However, attempting to use this command from your terminal to generate the key hash will require the JDK to be installed separately from Android Studio. If you are on a Mac, you can get around this requirement by using the `keytool` command in the context of Android Studio like this:

```
/Applications/Android\
Studio.app/Contents/jre/jdk/Contents/Home/bin/keytool -exportcert
-alias androiddebugkey -keystore ~/.android/debug.keystore |
```

```
openssl sha1 -binary | openssl base64
```

There are many more fields that will need to be filled out before you can make your Facebook application live (which you should come back and do at some point if you intend to release the application for production), but we can skip those for now.

Setting up Firebase

Now we will need to set up an application with Firebase, which we will be using for two main purposes:

- To handle authentication in our application (in conjunction with the Facebook API)
- To store data for our application and provide us with live updates

However, there are also many more features that Firebase provides that we won't be making use of.

We will handle the implementation of our database and authentication in our application later, for now, we just want to get everything set up that we need in our Firebase account. This will include creating a new project with Firebase and enabling Facebook as a sign-in method. We will eventually need to create a **Firestore** database in our Firebase account too, but we will be saving that for a later lesson.

To set up your Firebase project, you will need to follow the following steps:

1. Go to console.firebase.google.com and click **Add Project** (create an account first if you don't have one already)

2. Give your project a name and select a parent resource, then click **Continue**

3. Either turn analytics off or keep it on, and click **Create Project**

4. Once you are taken to the **Dashboard**, click the web icon `</>` above where it says **Add an app to get started**. If you are asked, you do not need to worry about setting up Firebase hosting (but you can if you want to). Give your application a nickname, and register the app.

5. After this step, you will be given your configuration code. We will not be using that code as is in our application, but we will need the details in the `config` object. Make note of this for later (or you can just come back to the dashboard again to get the configuration code). The config object will look something like this:

```
var firebaseConfig = {
  apiKey: "*************",
  authDomain: "*************",
  databaseURL: "*************",
  projectId: "*************",
  storageBucket: "",
  messagingSenderId: "*************",
  appId: "*************",
};
// Initialize Firebase
```

```
firebase.initializeApp(firebaseConfig);
```

You will notice that the full code supplied includes a `<script>` tag, but since we installed `firebase` through `npm` earlier we won't need to include that `<script>` tag. It is also worth noting that there is also another `npm` package available called @angular/fire that you can use to help integrate an Angular application with the various features Firebase offers. We will **not** be using this package. The main reasons we will not be using this package are that:

1. Our requirements are quite simple and will be straight-forward to implement ourselves
2. It serves as a better learning example to integrate the functionality ourselves rather than including an additional dependency to do it for us (you then might be able to use similar concepts to integrate other 3rd party libraries into your applications)
3. It is generally a good idea to avoid including additional dependencies (especially if they are large) in your application unless it is going to provide a significant benefit for the circumstances it is being used in
4. The more dependencies you use and the more layers you have, the more you are exposed to issues/bugs in those libraries.

There is nothing wrong with using additional libraries like this, but if we have simple requirements and can implement it easily enough ourselves, it is generally a good option to do so (I think it's better to under-use additional dependencies than it is to over-use them).

We will also need to enable **Facebook Authentication** inside of Firebase, to do this follow these steps:

1. Click on **Authentication** from the menu on the left side of the screen from the Firebase project dashboard
2. Click **Get started**
3. Enable **Facebook** as a sign-in method - click the edit button, and then toggle the enable switch and add your **App ID** and **App secret** from the Facebook application you created in the previous step.
4. Copy the **OAuth redirect URI** and add it to **Products > Facebook Login > Settings** under **Valid OAuth Redirect URIs** in your application in the Facebook developer portal. Click **Save Changes**.
5. Go back to the Firebase console and click **Save**.

Configuring the Native Projects

There are still a couple more steps we need to complete - we will need to add our **App ID** and **App Name** for the Facebook Application that we just created to our native iOS and Android projects, along with some other values. Let's start off with the Android project. You can open up the project in Android Studio by running:

```
ionic cap open android
```

Once the project is open, you will need to open the following file in Android Studio:

```
android > app > src > main > res > values > strings.xml
```

Inside of this file, you will need to add the following entries **inside** of the opening and closing `<resources>` tag:

```
<string name="fb_app_id">YOUR APP ID</string> <string name="fb_app_name">YOUR APP NAME</string>
```

Make sure to replace **YOUR APP ID** with your Facebook **App ID** and **YOUR APP NAME** with your Facebook **App Name**. Now we need to do something similar for iOS. Open the iOS project with:

```
ionic cap open ios
```

Once the project is open, you should:

1. Click **App** from the left menu (above **Pods**, select it don't expand it)
2. In the right panel, click the **Info** tab
3. Under **Custom iOS Target Properties** hover over any of the existing **Keys** and click the little **+** icon to add a new key
4. Create a new entry for **FacebookDisplayName**, set it to **String**, and add your Facebook App Name as the value
5. Create a new entry for **FacebookAppID**, set it to **String**, and add your Facebook App ID as the value
6. Create a new entry for **FacebookAutoLogAppEventsEnabled**, set it to **Boolean**, and set the value to `NO`

7. Create a new entry for **FacebookAdvertiserIDCollectionEnabled**, set it to **Boolean**, and set the value to `NO`
8. You will need to add an array of values under the **LSApplicationQueriesSchemes** key, add the key and set it to **Array**, and then add the following items under that array (by first clicking to expand the key - the arrow should point down - and then right click > Add Row):

▼ LSApplicationQueriesSchemes	Array	(4 items)
Item 0	String	fbshareextension
Item 1	String	fb-messenger-api
Item 2	String	fbapi
Item 3	String	fbauth2

The items in the image above are:

- Item 0 - String - fbshareextension
- Item 1 - String - fb-messenger-api
- Item 2 - String - fbapi
- Item 3 - String - fbauth2

You will also need to add a new **URL Scheme** that reflects your **Facebook App ID** prefixed with **fb**, like this:

Just scroll down to **URL Types** (make sure to expand it) and click the **+** button at the

bottom to add a new URL scheme. You will need to put the `fbYOUR-APP-ID` value in the **URL Schemes** field. If you do not do this, you will get an error complaining about not having the URL scheme registered:

```
Terminating app due to uncaught exception
'InvalidOperationException', reason: 'fb123456789 is not
registered as a URL scheme.
```

Finally, in order to correctly pass the authentication information back to our application, we will need to make some changes to the `AppDelegate.swift` file in the native iOS project. We won't be creating any new functions, we will just be modifying a couple of existing functions. First, we will need to import `FBSDKCoreKit` at the top of the `AppDelegate.swift` file:

```
import UIKit
import Capacitor
import FBSDKCoreKit
```

Now we need to add some code to some of the functions in this file. To make it clear what you need to add, I have surrounded the modifications with `// -- Added for Facebook Login`. Only add these sections to the existing functions, do not paste the entire function. It can be difficult to track down where exactly to add the code since there are so many functions all called `application`. I would recommend searching for the

existing comment `Override point for customization after application launch` to find the first function, and `Called when the app was launched with a url` for the second function.

```
  func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // Override point for customization after application launch.
    // -- Added for Facebook Login
    ApplicationDelegate.shared.application( application,
didFinishLaunchingWithOptions: launchOptions )
    // -- Added for Facebook Login
    return true
  }
```

```
  func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
    // Called when the app was launched with a url. Feel free to
add additional processing here,
    // but if you want the App API to support tracking app url
opens, make sure to keep this call
    // -- Added for Facebook Login
    ApplicationDelegate.shared.application( app, open: url,
sourceApplication:
```

```
options[UIApplication.OpenURLOptionsKey.sourceApplication] as?
String, annotation:
options[UIApplication.OpenURLOptionsKey.annotation] )
    // -- Added for Facebook Login
    return CAPBridge.handleOpenUrl(url, options)
  }
```

If you do not make these modifications, the login process will not complete properly after the user has finished authenticating using the Facebook app (when you go back to the application, the web based browser authentication window will remain visible).

Setting up Authentication

That was quite a lot of configuration, but we have everything we need set up now (until we get to dealing with storing our messages in the Firestore database, at least). Now we are going to spend some time back in our application. We will be setting up our `Auth` service to handle logins, and we will also wire up our pages to facilitate logging in and out as well.

The `Auth` service is going to be doing most of the logic and work for us, so let's get that implemented first. This will be a rather large service, so we will start off with a basic outline of the service, and then we will implement it bit by bit.

> Modify `src/app/services/auth.service.ts` to reflect the following:

```
import { Injectable, NgZone } from "@angular/core";
```

```
import { Platform } from "@ionic/angular";

import { Storage } from "@ionic/storage";
import { Facebook, FacebookLoginResponse } from "@ionic-
native/facebook/ngx";
import { User } from "../interfaces/user";

import { BehaviorSubject, Observable } from "rxjs";
import firebase from "firebase/app";
import "firebase/auth";

@Injectable({
  providedIn: "root",
})
export class AuthService {
  private loggedIn$: BehaviorSubject<boolean> = new
BehaviorSubject<boolean>(false);
  private user: User;

  constructor(
    private platform: Platform,
    private zone: NgZone,
    private facebook: Facebook,
    private storage: Storage
  ) {}

  public getUser(): User {
    return this.user;
  }
```

```
public getLoggedIn(): Observable<boolean> {
  return this.loggedIn$;
}

public init(): void {
  // Your web app's Firebase configuration
  const firebaseConfig = {
    apiKey: "***REPLACE****",
    authDomain: "***REPLACE****",
    databaseURL: "***REPLACE****",
    projectId: "***REPLACE****",
    storageBucket: "",
    messagingSenderId: "***REPLACE****",
    appId: "***REPLACE****",
  };

  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);

  // Emit logged in status whenever auth state changes
  firebase.auth().onAuthStateChanged((firebaseUser) => {
    this.zone.run(async () => {
      if (firebaseUser) {
        this.user = {
          uid: firebaseUser.uid,
          displayName: firebaseUser.displayName,
          displayPicture: firebaseUser.photoURL,
        };
```

```
          const accessToken = await this.storage.get("fb-access-
token");
          if (accessToken) {
            this.setAccessTokenForPhoto(accessToken);
          }

          this.loggedIn$.next(true);
        } else {
          this.user = null;
          this.loggedIn$.next(false);
        }
      });
    });
  }

  public login(): void {}

  public async logout(): Promise<void> {}

  private async nativeFacebookAuth(): Promise<void> {}

  private async browserFacebookAuth(): Promise<void> {}

  private isUserEqual(
    facebookResponse: FacebookLoginResponse,
    firebaseUser: firebase.User
  ): boolean {
    return true;
```

```
  }

  private setAccessTokenForPhoto(accessToken: string): void {
    // Update display picture with non-default facebook image
    if (this.user) {
      this.user.displayPicture = `${this.user.displayPicture}?
access_token=${accessToken}`;
    }
  }
}
```

IMPORTANT: Replace `firebaseConfig` with the values from your own configuration code that you received from Firebase earlier.

We actually have some interesting imports in this service, so let's talk through some of those. We are importing `NgZone` which will allow us to force code to run inside of Angular's "zone". Since we will be utilising events originating from the Firebase Javascript SDK, which is outside of Angular's "zone", code triggered as a result of these events might not be picked up by Angular's change detection (which would update our user interface in response to changes). To solve this issue, we will use `NgZone` to manually run parts of our code inside of Angular's zone.

We are also importing `BehaviorSubject` which allows us to create a special type of observable. You can `subscribe` to a `BehaviorSubject` just like you can with a normal observable, but one key different is that a `BehaviorSubject` has a default value and will always return some value. We will be using this to determine whether or not a user is

currently logged in, and we can also `subscribe` to it to be notified when the authentication status of the user changes.

The other two interesting imports we have are `firebase` and `firebase/auth` - these provide us with access to the functionality of the Firebase Javascript SDK, and the `firebase/auth` import gives us access to the authentication functionality of Firebase.

We then have our class member variables:

```
  private loggedIn$: BehaviorSubject<boolean> = new
BehaviorSubject<boolean>(false);
  private user: User;
```

We set up our `BehaviorSubject` on an instance variable called `loggedIn$` - this originally will have a value of `false`, but later we will emit a `true` value once the user has logged in. Our `user` variable will just hold a reference to the user's information (as defined in our `User` interface that we created earlier).

We have implemented the `init` method entirely already. It is important that you replace the config object with your own values from the Firebase project that we just set up recently. We initialise Firebase by calling the `initializeApp` method with our config object. We also set up a listener for `onAuthStateChanged()` which will trigger every time the authentication state of the user changes with Firebase (e.g. when they log in or log out). If `firebaseUser` is truthy, then the user is logged in. We set the `user` object, and trigger `next` on `this.loggedIn` which will cause our `BehaviorSubject` to emit `true`

to anything that is subscribed to it. If `firebaseUser` is falsy, it means the user is logged out, and so we set the `user` object to null and emit a `false` value.

Notice that this code, which is triggered by Firebase, is forced to run inside of Angular's zone by using the `run()` method. Also notice that we are calling a `setAccessTokenForPhoto` method, which has also already been implemented in the code above. Facebook now requires that the `accessToken` is appended to the URL when trying to get a user's profile photo, otherwise we will just get the default Facebook profile picture. This causes us a little bit of trouble, because the `accessToken` is only given to us after the initial login (not when the user is automatically re-authenticated later). This means we need to store a reference to this `accessToken` in storage, which we do by using Ionic's Storage API.

The remaining methods in this service, which we will implement fully in a moment, are as follows:

```
public login(): void {}

public async logout(): Promise<void> {}

private async nativeFacebookAuth(): Promise<void> {}

private async browserFacebookAuth(): Promise<void> {}

private isUserEqual(
  facebookResponse: FacebookLoginResponse,
  firebaseUser: firebase.User
```

```
  ): boolean {
    return true;
  }
```

We have a `login` and `logout` method which is reasonably obvious, as well as two methods that provide different authentication methods: `nativeFacebookAuth()` and `browserFacebookAuth()`. Our `login` function will automatically determine whether the application is running on the web or natively. If it is running on the web it will use `browserFacebookAuth()` and if it is running natively it will use `nativeFacebookAuth()`. The `isUserEqual` method will just be a helper method that is provided by Firebase to help with the authentication.

> Modify the `login` function to reflect the following:

```
  public login(): void {
    if (this.platform.is("capacitor")) {
      this.nativeFacebookAuth();
    } else {
      this.browserFacebookAuth();
    }
  }
```

This method is quite straight-forward. We just use the **Platform API** to detect what platform the application is running on. If it is `capacitor` (e.g. it is running on iOS or

Android) we trigger the `nativeFacebookAuth()` method, otherwise we trigger `browserFacebookAuth()`.

> Modify the `logout` function to reflect the following:

```
public async logout(): Promise<void> {
  if (this.platform.is("capacitor")) {
    try {
      await this.facebook.logout(); // Unauth with Facebook
      await firebase.auth().signOut(); // Unauth with Firebase
    } catch (err) {
      console.log(err);
    }
  } else {
    try {
      await firebase.auth().signOut();
    } catch (err) {
      console.log(err);
    }
  }
}
```

The basic idea with this function is to log the user out. If we are running the application natively it has to logout both with the native Facebook plugin and Firebase, otherwise, just the Firebase sign out needs to be triggered.

This code is actually a good demonstration of where wrapping a `void` method in a Promise might be useful. We don't necessarily need to wait for anything in this method, but both the Facebook and Firebase sign out methods are asynchronous (e.g. they do not happen instantly, and complete after a small amount of time). We `await` both of these functions, meaning the Promise that this method is wrapped in won't complete until anything marked with `await` has completed. If we were to call this method from elsewhere, we could `await` it, to ensure that the logout operations have all completed before doing something, e.g:

```
async someFunction(){
  await this.authService.logout(); // wait for the logout
operation to complete
  // now do something else
}
```

You might not need to wait for an assurance that the logout operation has completed, but it does give you the option if you need.

> Modify the `nativeFacebookAuth` function to reflect the following:

```
  private async nativeFacebookAuth(): Promise<void> {
    try {
      const facebookResponse = await
this.facebook.login(["public_profile", "email"]);
```

```
this.setAccessTokenForPhoto(facebookResponse.authResponse.accessTok

      this.storage.set("fb-access-token",
facebookResponse.authResponse.accessToken);

      if (facebookResponse.authResponse) {
        // User is signed-in Facebook.
        const unsubscribe =
firebase.auth().onAuthStateChanged((firebaseUser) => {
          unsubscribe();
          // Check if we are already signed-in with Firebase with
the correct user.
          if (!this.isUserEqual(facebookResponse, firebaseUser)) {
            // Build Firebase credential with the Facebook auth
token.
            const credential =
firebase.auth.FacebookAuthProvider.credential(
              facebookResponse.authResponse.accessToken
            );
            // Sign in with the credential from the Facebook user.
            firebase
              .auth()
              .signInWithCredential(credential)
              .catch((error) => {
                console.log(error);
                this.loggedIn$.next(false);
              });
          } else {
```

```
            // User is already signed-in Firebase with the correct
user.
            console.log("already signed in");
          }
        });
      } else {
        // User is signed-out of Facebook.
        firebase.auth().signOut();
      }
    } catch (err) {
      console.log(err);
      this.loggedIn$.next(false);
    }
  }
```

This is the big one, and comments have been included above to help explain the process. This is, more or less, just the standard authentication code provided by Firebase - with a bit of a twist. This code allows us to manually pass in Facebook's `authResponse` to Firebase to authenticate the user with Facebook (rather than triggering Facebook authentication through Firebase itself). The reason we do this is so that we can instead trigger our native Facebook plugin to get this authentication information. This line:

```
const response = await this.facebook.login(["public_profile",
"email"]);
```

triggers the `cordova-plugin-facebook4` plugin that we installed and set up. Since we have already imported **Facebook** from Ionic Native we can access it through the `this.facebook` class member that we set up through the constructor. We call the `login` method and pass in an array of permissions we require. We're only after basic user information so we only request permission for the users "public profile" and "email", but there are other types of permissions you can request. Take a look at [this page]() for a full list.

This will authenticate the user using the native Facebook application (if it is installed) and then pass the value back to our application. We then pass that value to Firebase to complete the authentication on their end.

Notice that Firebase is making use of the `isUserEqual` method here to check if the user is already signed in with Firebase - let's implement that function now (this is just code provided by Firebase, with some minor modifications).

> Modify the `isUserEqual` function to reflect the following:

```
private isUserEqual(
  facebookResponse: FacebookLoginResponse,
  firebaseUser: firebase.User
): boolean {
  if (firebaseUser) {
    const providerData = firebaseUser.providerData;

    providerData.forEach((data) => {
      if (
        data.providerId ===
```

```
firebase.auth.FacebookAuthProvider.PROVIDER_ID &&
          data.uid === facebookResponse.authResponse.userID
        ) {
          // We don't need to re-auth the Firebase connection.
          return true;
        }
      });
    }

    return false;
  }
```

> Modify the `browserFacebookAuth` function to reflect the following:

```
  private async browserFacebookAuth(): Promise<void> {
    const provider = new firebase.auth.FacebookAuthProvider();

    try {
      const result: any = await
firebase.auth().signInWithPopup(provider);
      this.setAccessTokenForPhoto(result.credential.accessToken);
      this.storage.set("fb-access-token",
result.credential.accessToken);
      console.log(result);
    } catch (err) {
      this.loggedIn$.next(false);
```

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

Finally, we come to our last method, which is the browser authentication method. This one is a bit simpler because we are just using the standard Firebase authentication flow, which will pop up a window for authenticating with Facebook. Notice how we don't actually need to return any confirmation of the user logging in, we have our `onAuthStateChanged()` listener set up which will automatically trigger whenever the users authentication state changes (which will then also trigger our `loggedIn` observable).

Using Authentication

Now that our `Auth` service is set up, we can make use of it in our application. Let's start by making the `login` functionality on our login page actually work. First, we will implement the method that kicks off the login process.

> Modify the `login` method in `src/app/pages/login/login.page.ts` to reflect the following:

```
login() {
  this.showLoading();
  this.authService.login();
}

async showLoading() {
```

```
    this.loading = await this.loadingCtrl.create({
      message: "Authenticating...",
    });
    this.loading.present();
  }
```

Before we do anything, we create a loading overlay which we display whilst we are performing the Facebook authentication. Some stuff needs to happen behind the scenes before we can let the user proceed, and we want them to know that the login is in process. To handle this, we have created a separate `showLoading` method.

All we need to do to trigger a login is to call the `login` method from the `AuthService`. The way in which we detect whether or not the log in is successful comes from our `loggedIn` observable that the `Auth` service also provides. What we can do is `subscribe` to data that is being emitted from that observable to determine when to send the user to the logged-in view.

> Modify the `ngOnInit` method in `src/app/pages/login/login.page.ts` to reflect the following:

```
  async ngOnInit() {
    await this.showLoading();

    this.authService
      .getLoggedIn()
      .pipe(debounceTime(1500))
```

```
      .subscribe((isLoggedIn) => {
        this.loading.dismiss();

        if (isLoggedIn) {
          this.menu.enable(true);
          this.navCtrl.navigateForward("/home");
        }
      });
  }
```

This subscribes to `loggedIn` and checks the `status` which will indicate if the user has successfully logged in. If that is the case, we enable menu interaction and then we send them to the `/home` route. In either case, we `dismiss` the loading overlay because the authentication process has completed (it will have either succeeded or failed). We also trigger `showLoading()` at the beginning of this method, because Firebase will immediately attempt to authenticate the user before they manually click the login button. If the user has previously authenticated, then they will be immediately taken through to the logged-in view, otherwise, the loading overlay will just be dismissed as usual and the user can trigger the Facebook login themselves.

Notice that we `pipe` a `debounceTime` operator on here. This will cause our observable to wait for our value to be "steady" (e.g. it hasn't changed) for `1.5` seconds. The reason we do this is because our `getLoggedIn()` will always give us a value immediately which will cause our loading overlay to be dismissed, even if the authentication is still happening. By using `debounceTime` we can give the authentication time to run first.

We have already set up the `logout` method for the `Auth` service, but we do need to trigger that at some point as well. Our logout button will be in our side menu, which has its template in the root component, so we will need to trigger it from there.

> Modify the logout method in `src/app/app.component.ts` to reflect the following:

```
async logout() {
  await this.menu.close();
  this.authService.logout();
}
```

First, we close the side menu, and then we simply call the `logout` method in the `Auth` service. This will trigger our `loggedIn` observable to eventually emit a `false` value, and we will eventually add logic into our other pages to send the user back to the login page when this happens.

There is one more important step we need to take here, and that is to call the `init` method of our `Auth` service at some point during our application start up. We will also add this to our root component so that it is triggered right away.

> Modify `src/app/app.component.ts` to reflect the following:

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
```

```
import { MenuController } from "@ionic/angular";
import { AuthService } from "./services/auth.service";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(private authService: AuthService, private menu:
MenuController) {
    this.initializeApp();
  }

  initializeApp() {
    this.authService.init();

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#54a0ff" }).catch((err)
=> {
      console.warn(err);
    });
  }
```

```
  async logout() {
    await this.menu.close();
    this.authService.logout();
  }
}
```

Protecting Routes

Since navigation in our application is based on the URL, the user could easily bypass the login screen just by manually typing `/home` into the URL bar. This is less of an issue when the application is running natively, and the URL bar isn't easily accessed, but we will walk through a way to prevent this to provide a better experience on the web.

Keep in mind that what we are about to do is all on the client-side, and anything on the client-side can easily be worked around by nefarious users. The main purpose of implementing something like this is to keep your honest users on a sensical path within your application. If certain pages in your application contain sensitive/secret content, the best way to protect that content is by storing it on a server and only loading it into the application once the user has supplied valid credentials.

The key point is that what we are about to do is for user experience, not security.

With that disclaimer out of the way, let's get into it. We can add a `canActivate` property to any route in our application which will allow us to programmatically control when it can be navigated to. We are going to create a service called `AuthGuardService` which we will supply to this property.

> Modify `src/app/services/auth-guard.service.ts` to reflect the following:

```
import { Injectable } from "@angular/core";
import { Router } from "@angular/router";
import { AuthService } from "./auth.service";

@Injectable({
  providedIn: "root",
})
export class AuthGuardService {
  constructor(private router: Router, private authService:
AuthService) {}

  canActivate(): boolean {
    if (!this.authService.getUser()) {
      this.router.navigate(["login"]);
      return false;
    }

    return true;
  }
}
```

The basic idea is that we implement a `canActivate` method. If that method returns `true` then the navigation will be allowed, if it returns `false` it won't be - and we can run whatever other code we like. In this case, if the `user` object is `null` or `undefined`, then

we want to kick them back to the `login` route. Otherwise, we are happy for the navigation to proceed as normal.

Now, all we need to do is attach that service to the routes we want to protect.

> Modify `src/app/app-routing.module.ts` to reflect the following:

```
import { NgModule } from "@angular/core";
import { PreloadAllModules, RouterModule, Routes } from
"@angular/router";
import { AuthGuardService } from "./services/auth-guard.service";

const routes: Routes = [
  {
    path: "",
    redirectTo: "home",
    pathMatch: "full",
  },
  {
    path: "login",
    loadChildren: () => import("./login/login.module").then((m) =>
m.LoginPageModule),
  },
  {
    path: "home",
    loadChildren: () => import("./home/home.module").then((m) =>
m.HomePageModule),
```

```
    canActivate: [AuthGuardService],
  },
  {
    path: "about",
    loadChildren: () => import("./about/about.module").then((m) =>
m.AboutPageModule),
    canActivate: [AuthGuardService],
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Notice that we have imported the service, and we have supplied it to the `canActivate` property for the routes we want to protect. The `canActivate` property accepts an array of values, so you can add more than one service here if you like.

That's all there is to the Facebook integration in this application, but there's a ton more you can do with the Facebook API. I'd recommend looking at the plugin documentation and the Facebook Graph API to get more of a sense of what is possible with Facebook.

In the next lesson, we'll be looking at how we can add some messages to our chat screen!

Lesson 5: Creating Messages & Navigation

The interface for sending messages is obviously one of the most critical bits of functionality in our application, but it's going to be pretty straightforward to implement. The hard part will be the integration with Firebase which we will tackle in the next lesson. For now, we just want to create the ability for the logged in user to add messages to the screen.

We will also be tieing up a few loose ends with our navigation - we have a menu with some options that do nothing and an about page that has nothing on it.

Adding Messages

We've already set up an input field tied to the `this.chatMessage` variable in our home page class definition and a button that triggers the `sendMessage` function, so all we have to do to enable creating a message is to define that `sendMessage` function. We'll also have to modify the template to loop through and display all of the available messages.

> Modify the `sendMessage` function in `src/app/home/home.page.ts` to reflect the following:

```
  async sendMessage() {
    if (this.chatMessage.trim().length > 0 && !this.addingMessage)
{
```

```
      this.addingMessage = true;
      await this.messagesService.addMessage(this.chatMessage);
      this.addingMessage = false;
      this.chatMessage = "";
    } else {
      this.chatMessage = "";
    }
  }
```

We are passing along the `chatMessage` to the `addMessage` method of our `Message` service (which we haven't actually implemented yet). This will eventually handle uploading that message to our Firestore database (again, which doesn't exist yet) and then we just reset the `chatMessage` back to a blank value so that the user can start typing their next message.

Notice that we set the `addingMessage` boolean to `true`, then we wait for our `addMessage` method to resolve, and then we set it back to `false`. This allows us to use this `boolean` to check if a message is in the process of being uploaded, if it is we don't allow a new message to be submitted.

NOTE: Your application will trigger an error if you attempt to serve it before implementing the `addMessage` method.

We will also need to modify the template so that it is capable of displaying the messages.

> Modify the list in `src/app/home/home.page.html` to reflect the following:

```
  <ion-list lines="none">
    <ion-item *ngFor="let message of messages">
      <ion-avatar
        [attr.slot]="message.displayName == user.displayName ?
'start' : 'end'"
      >
        <img [src]="message.displayPicture" />
      </ion-avatar>
      <div class="chat-message">
        <ion-note>{{ message.displayName }}</ion-note>
        <p>{{ message.message }}</p>
      </div>
    </ion-item>
  </ion-list>
```

Now we're looping through and displaying all of our messages (which will eventually be contained in the `messages` member variable on the home page) in their own `<ion-item>`. Remember that the star syntax is a shortcut for creating embedded templates, and `*ngFor` will create an embedded template for every individual `message` in `messages` and stamp it out with that specific messages values. Using the data that is attached to each message we are now adding the users display picture and name to the message, along with the message itself.

We mostly have a pretty basic template set up, but we are doing something a little funky:

```
<ion-avatar
  [attr.slot]="message.displayName == user.displayName ? 'start' :
'end'"
>
```

As a way to denote which messages are owned by the logged in user, we are going to switch the side that the avatar is displayed on from left to right. What we want to do is apply `slot="start"` to the `<ion-avatar>` if the current user's username matches the username of the message, and `slot="end"` if it doesn't. We can achieve this by conditionally setting `attr.slot` based on the result of our little if condition (we are using the ternary syntax which is basically: `condition ? do this if true : do this if false`).

This is all we need to do with this page for now, but we won't actually be able to use this functionality until we finish implementing the `Message` service in the next lesson.

Creating the About Page

We've got our sliding menu set up, and we are linking to the **About** page, but we haven't built it yet. Let's change that.

> Modify `src/app/about/about.page.html` to reflect the following:

```
<ion-header>
  <ion-toolbar color="primary">
```

```
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>
      <img src="assets/images/logo.png" />
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-card>
    <ion-card-content>
      <p>
        Camper Chat allows you to talk to other
        <strong>caravaners, RV'ers, road trippers and
campers</strong> near you. Use Camper Chat to
        give and receive tips, ask for help with your flat tyre,
or just have a friendly chat.
      </p>
    </ion-card-content>
  </ion-card>
</ion-content>
```

This is a pretty unimportant page in the context of our application, but we do need to make sure that if the `logout` method is triggered whilst on this page that it will send the user back to the **Login** page.

> Modify `src/app/about/about.page.ts` to reflect the following:

```
import { Component, OnInit, OnDestroy } from "@angular/core";
import { NavController } from "@ionic/angular";
import { Subscription } from "rxjs";
import { AuthService } from "../services/auth.service";

@Component({
  selector: "app-about",
  templateUrl: "./about.page.html",
  styleUrls: ["./about.page.scss"],
})
export class AboutPage implements OnInit, OnDestroy {
  private loggedInSubscription: Subscription;

  constructor(private navCtrl: NavController, private authService:
AuthService) {}

  ngOnInit() {
    this.loggedInSubscription =
this.authService.getLoggedIn().subscribe((status) => {
      if (!status) {
        this.navCtrl.navigateBack("/login");
      }
    });
  }

  ngOnDestroy() {
    this.loggedInSubscription.unsubscribe();
```

```
  }
}
```

We are listening for updates to that `loggedIn` observable, and if we detect that the user is now unauthenticated we send them back to the login page. Now the user can switch back and forth freely between the about page and the chat page, and then they can also log out to go back to the login page.

Remember that the application will not actually compile properly now as we are referencing a method in the `Message` service that does not yet exist. We will fix that in the next lesson.

We've now got all the basic functionality of the application set up, the last big thing we need to tackle is the integration of our Firestore database, which we will be doing in the next lesson, and then we just need to tidy things up and make it look a bit nicer.

Lesson 6: Using Firestore to Store Messages

Although we've already created most of the functionality for the application, this lesson is going to be the most significant of them all. In this lesson, we are going to enable the saving of our user's messages to our backend as well as facilitating live data updates.

Firestore is a document-based NoSQL database, and in order to properly understand how to use it we will need to cover some basic concepts. Our use of the database will be rather simple, so it is not critical that you have a thorough understanding of how a document-based database works, but it will certainly help to understand at least a little.

A Brief Introduction to Document-Based NoSQL Databases

Before we get into specifically discussing "document" databases, I think it is useful to quickly cover some other types of NoSQL databases (which basically covers anything that is not a typical "relational" style database like MySQL). Some types of NoSQL databases include:

- Key-Value (redis, Amazon DynamoDB, memcached)
- Columnar / BigTable (HBase, Cassandra, Amazon SimpleDB)
- Document (Firestore, CouchDB, MongoDB, Riak)
- Graph (Neo4J, AllegroGraph, InfoGrid)

Two commonly used NoSQL database structures are Key-Value and Document (like the Firestore database we will be using). Key-Value and Document-based databases are both very similar in nature, but there are differences.

A Key-Value database is a simple way to store values indexed by a `key`. You would already be familiar with this concept if you have used the browsers local storage to store data. To use the browsers local storage as an example, you would set a value on a key like this:

```
localStorage.setItem("name", "Josh");
```

and then you could later retrieve the value by using the key like this:

```
localStorage.getItem("name");
```

A document database is similar but stores its information in a structure that closely resembles Javascript objects or JSON strings. These "objects" that are stored are referred to as "documents". I think this terminology is quite confusing because as a beginner when you think of a "document database" you would probably be thinking in the context of documents like a word document or spreadsheet. This isn't the case though, a document in this context is simply a JSON object like this:

```
{
    "id": 1,
    "name": 'Josh',
    "country": 'Australia',
    "interests": ['Ionic', 'Writing', 'Travelling']
```

```
}
```

The reason these are referred to as "documents" is because a document generally contains all the information you need in one place. If I look at an invoice, I could see all the data I need right there: `biller`, `items`, `subtotal`, `tax`, and so on. In a document based NoSQL database this might look like this:

```
{
    "id": 392,
    "biller": 'Bob\'s Building Supplies',
    "items": ['bricks', 'paint', 'wrench'],
    "subtotal": 230.32,
    "tax": 0
}
```

In a relational database, this information might be split up across multiple tables (not all in one place like a "document").

You could also store a similar JSON object as a value in a Key-value database and then retrieve that entire JSON object later. So, what's the difference between the two approaches? The difference is that with a document database the database knows about the structure of that JSON object, and provides a means to query against it, rather than just directly accessing a specific key value. If the above was just a chunk of data we needed to retrieve we could easily store it in a Key-value database, but if we wanted to run a query to return objects where the country was 'Australia', then it would be better to use a

document-based database.

This is a very brief introduction to the concept of a document-based database, and if you don't have any other experience with them you will likely need to do some additional learning to be able to build your own databases. Fortunately, our requirements for this application are quite simple and involve just storing a single "collection" of "documents" called "messages".

Creating a Firestore Database

To enable Firestore, you will need to click on the **Cloud Firestore** option inside of your Firebase dashboard (on the left-hand side menu). You should then see an option like this:

Just click on **Create database**. You will then be given the option to create this database in **locked mode** or **test mode**. This will define the initial set of security rules for your database that determine who can access the data in the database and how.

IMPORTANT: Make sure to choose the **test mode** option.

We will be using **test mode** for now, and we will implement proper security rules later on.

This will allow us to interact with the database with no restrictions, but it is important that you **do not deploy to production with test mode**. Essentially, being in test mode means that anybody can do anything to the database. Keep in mind that Firestore will automatically disable read/write access to test databases after 30 days, so if you are coming back to this example after a while you may need to skip ahead to implement the security rules in order to continue accessing the database.

Select **Start in test mode** and then click **Enable**.

Your Firestore database is now ready to use! We will be managing data completely from within the application, but you can also see (and modify) the data for your application from this interface. We will only need to come back to this interface later to set up our security rules.

Implementing the Message Service

Now we get to the fun bit. This is where we will pull in data from our Firestore database, as well as setting up the `addMessage` method so that users can add new messages to the database. This is a significant bit of functionality, but you might be pleased to see that it is actually quite short.

Let's once again implement an outline of this service first, and then we will implement each function individually.

> Modify `src/app/services/message.service.ts` to reflect the following:

```
import { Injectable, NgZone } from "@angular/core";

import { BehaviorSubject } from "rxjs";
import firebase from "firebase/app";
import "firebase/firestore";

import { AuthService } from "./auth.service";
import { Message } from "../interfaces/message";
import { FirebaseFirestore, DocumentReference } from
"@firebase/firestore-types";

@Injectable({
  providedIn: "root",
})
export class MessagesService {
  private db: FirebaseFirestore;
  public messages: BehaviorSubject<Message[]> = new
BehaviorSubject<Message[]>([]);

  constructor(private zone: NgZone, private authService:
AuthService) {}

  init(): void {
    this.db = firebase.firestore();
  }

  watchMessages(): Function {}
```

```
  async addMessage(message: string): Promise<DocumentReference> {}
}
```

A lot of the basic outline for this service is quite similar to our `Auth` service, and we will be using a similar concept with the `BehaviorSubject`, except this time it will be emitting messages from our database instead of the logged-in status. Also, notice that we are now importing `firebase/firestore` instead of `firebase/auth`.

We have also set up a class member of `db` which will hold a reference to our Firestore database once the `init` method has been called.

We then just have two methods: `watchMessages()` and `addMessage()`. The `watchMessages()` function will be responsible for triggering our `messages` observable with any new messages that are added to the Firestore database. The `addMessage` method will handle adding a users message to the Firestore database.

> Modify the `watchMessages` function to reflect the following:

```
  watchMessages(): Function {
    this.messages.next([]);

    return this.db
      .collection("messages")
      .orderBy("created", "desc")
      .limit(50)
```

```
      .onSnapshot(querySnapshot => {
        let messages = [];

        querySnapshot.docChanges().forEach(change => {
          if (change.type === "added") {
            messages.push({
              ...change.doc.data()
            });
          }
        });

        this.zone.run(() => {
          this.messages.next(messages);
        });
      });
  }
```

Firestore provides us with the ability to query our database, which we do by using the various methods that we have chained onto `this.db`. We are saying that we are interested in the `messages` collection of documents, we want them ordered by their `created` field in descending order, and we want to retrieve at most `50` of them from the database. The `onSnapshot` will trigger initially, and every time that new data is added. When this happens, we loop through the data that is sent and we add any new documents (determined by `change.type` being `added`) to a `messages` array. We then pass that array of messages to our `BehaviorSubject` which will emit the messages data to anything that is subscribed to it. We will make use of that in our home page later in this lesson to display the message data.

An important aspect of this method is that by returning our call to `this.db.collection...` we will have access to a function that can be used to stop listening for data updates (we will utilise this later).

> Modify the `addMessage` function to reflect the following:

```
async addMessage(message: string): Promise<DocumentReference> {

  const user = this.authService.getUser();

  return await this.db.collection("messages").add({
    uid: user.uid,
    displayName: user.displayName,
    displayPicture: user.displayPicture,
    message: message,
    created: Date.now()
  });
}
```

This method just takes in a message, and then adds that to the database by using the `add` method. We pass along all the data we want to include in the document, which includes the user's id, name, picture, the message itself, and the current date/time.

Just like with our `Auth` service, we will also need to make sure that we call the `init` method during the application startup.

> **Modify `src/app/app.component.ts` to reflect the following:**

```
import { Component } from "@angular/core";
import { Plugins } from "@capacitor/core";
import { MenuController } from "@ionic/angular";
import { AuthService } from "./services/auth.service";
import { MessagesService } from "./services/messages.service";

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: "app-root",
  templateUrl: "app.component.html",
  styleUrls: ["app.component.scss"],
})
export class AppComponent {
  constructor(
    private authService: AuthService,
    private messagesService: MessagesService,
    private menu: MenuController
  ) {
    this.initializeApp();
  }

  initializeApp() {
    this.authService.init();
    this.messagesService.init();
```

```
    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.setBackgroundColor({ color: "#54a0ff" }).catch((err)
=> {
      console.warn(err);
    });
  }

  async logout() {
    await this.menu.close();
    this.authService.logout();
  }
}
```

Using the MessageService

Now that we have our `Message` service implemented, we need to make use of it on our home page to display some messages (and also make sure that we display any new messages that come in).

We have already implemented most of the home page, we just need to finish setting up our `ngOnInit()` and `ngOnDestroy()` lifecycle hooks.

> Modify `ngOnInit` in `src/app/home/home.page.ts` to reflect the following:

```
  ngOnInit() {
    this.mutationObserver = new MutationObserver((mutations) => {
      setTimeout(() => {
        this.contentArea.scrollToBottom(200);
      }, 0);
    });

    this.mutationObserver.observe(this.chatList.nativeElement, {
      childList: true,
    });

    this.loggedInSubscription =
this.authService.getLoggedIn().subscribe(async (loggedIn) => {
      if (loggedIn) {
        this.user = this.authService.getUser();
        this.detachListener =
this.messagesService.watchMessages();
        this.messagesSubscription =
this.messagesService.messages.subscribe((messages) => {
          messages.reverse();
          messages.forEach((message) => {
            this.messages.push(message);
          });
        });
      } else {
        this.navCtrl.navigateBack("/login");
      }
    });
```

```
  }
```

We are also subscribing to the `loggedIn` observable from our `Auth` service here, and if the user is not authenticated we send them back to the login screen. However, if the user is authenticated we trigger our Firestore functionality. We first trigger the `watchMessage` method which will cause our `messages` observable to start emitting the messages data. We want to display these messages in reverse order (newest messages at the bottom of the list) so we first call the `reverse()` method, and then we push each of the messages into our messages array that is being used to display the list. Every time new data is added to the database, it will get pushed into the list.

Perhaps the more interesting part here is the use of the "mutation observer" - remember when I said we were going to make use of those references we got with `@ViewChild` earlier, well, now is the time. A mutation observer basically just lets us watch an element for changes. Since we want to scroll the list to the bottom every time something is added inside of `<ion-list>` (because the new messages will be added at the bottom) this is a good option for us to trigger this behaviour.

First, we create a new mutation observer and set up its handler. Whenever a mutation is detected, we want to get the content area, and then trigger the `scrollToBottom` method. Note that we defer triggering the scroll until other tasks have finished by calling `scrollToBottom` asynchronously using a `setTimeout` with a timeout of `0`. Next, we tell the mutation observer what to watch. In this case, we are telling it to watch the `chatList` (which is our `<ion-list>`) and we supply `childList: true` so that it will trigger whenever there is changes to the children of `<ion-list>` (i.e. the `<ion-item>` elements).

> Modify `ngOnDestroy` in `src/app/home/home.page.ts` to reflect the following:

```
ngOnDestroy() {
  this.loggedInSubscription.unsubscribe();
  this.messagesSubscription.unsubscribe();
  this.detachListener();
}
```

The purpose of this method is to clean up our subscriptions once the home page is destroyed. We unsubscribe from both the `loggedIn` and `messages` observables, and we also use the function we returned from `watchMessages` to stop listening for updates from Firestore by calling `this.detachListener()`.

That's it! You should now have a fully functional chat application. It works, but it could look a bit nicer. In the next lesson, we are going to cover setting up Firestore security rules. After that, we are going to make the application look a bit prettier and even set up some animations!

Lesson 7: Firestore Security Rules

At this point, we have most of the functionality for our application completed. We can both add new chat messages to our Firestore database, and retrieve them so that we can display them in the application. However, when we initially created the database we created it in "test mode", which means that our database has no security whatsoever. Anybody would be able to read or modify anything they like in our database. This makes it easy to develop our application, but obviously, we do not want this in a production environment.

In order to combat this, we can implement **Firestore Security Rules**. These rules can be found in the **Rules** tab which is next to the **Data** tab inside of your Firestore database (accessed through your project in the Firebase console). The set of rules which we have currently (since we are in test mode) look like this:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2020, 12, 26);
    }
  }
}
```

This setup means that all **reads** and **writes** will be allowed to the database, no matter who is making them. If you wanted to change these rules, you would need to make modifications to them here and **publish** the new rules.

NOTE: Notice that a timestamp 30 days from whenever you created the database in test mode is included here. If you need additional development/play time with your database without creating proper security rules you could extend this time to a later date. But **do not** deploy a production application with these rules.

In this lesson, we are going to go on a bit of a journey from creating basic security rules to the final security rules that our application will require in order to be secured to the level that we need. Along the way, this will allow us to explain various aspects of how Firestore Security Rules work.

We won't be covering *every* aspect of Firestore Security Rules, there is just too much to learn, but this should give you a solid grounding for how creating security rules work in general. The exact kinds of rules you need to write will depend on the specifics of the application you are creating.

IMPORTANT: I will always recommend in cases where security is important to your application, that you should only handle it yourself if you have a solid understanding of how to secure your application and data. There are some cases where perhaps you are just building an application for fun, or when you aren't storing important data and any stolen data or data loss will not matter - in these cases, you might choose to tackle security yourself even if you do not have a strong understanding. However, there are many more situations where security is of critical importance, and a surface level understanding isn't enough to take on the responsibility yourself.

Despite my warning, please do attempt to learn and have fun with security in the right environment. Even if it may take a while to learn enough to start handling all security concerns yourself, even knowing a little bit can provide you with a lot of insight.

Match & Allow

Creating Firestore security rules mostly comes down to these two concepts: `match` and `allow`. We will use `match` to target a specific collection or document, and we will use `allow` to define an expression to determine how that specific collection/document may be accessed.

When writing our `match` statements, there is a bit of boilerplate code that mostly won't change. For example, the default rules for a "locked" database (the opposite of our "test mode" database) look like this:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

This might look a bit intimidating, but we can mostly forget about all of this:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

  }
}
```

This code just means that we are targeting the `firestore` service, and the first `match` statement just means we are targeting the default database in our project - we won't need to change this. All of the important stuff will be *inside* of this initial `match`, e.g:

```
    match /{document=**} {
      allow read, write: if false;
    }
```

This particular `match` uses the `{document=**}` recursive wildcard to target every document in the database, and in this case, it is blocking all reads and writes. You won't typically just use a blanket `match` like this, but rather target specific documents or collections with your `match` statement, e.g:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
```

```
    match /messages/{message} {
      allow read;
      allow write: if false;
    }

  }
}
```

In this case, we are attempting to `match` the `messages` collection. We use the `{message}` wildcard to match against any message in the collection, and then this will also make a `message` variable available to the `allow` expression we write (such that we could check the specifics of a document in our rule). You don't have to use a wildcard, you could also just match a single specific document, but you can't just match an entire collection by doing this: `/messages/`. If you want your security rules to apply to an entire collection, you must use a wildcard: `/messages/{message}`.

Although we are only matching one collection here, and this is all we will need for the application we are building, you can also match other collections or documents (or even sub-collections) and create different rules for those. For example, we might want to do something like this:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
```

```
    match /messages/{message} {
      allow read;
      allow write: if false;
    }

    match /posts/{post} {
      allow read;
      allow write: if false;

        match /comments/{comment} {
          allow read;
          allow create: if true;
          allow update, delete: if false;
        }

    }

  }
}
```

This would assume we have another collection called `posts` and within that, a sub-collection called `comments`. We can structure our security rules in this way to define access to different collections however we like.

Now let's briefly touch on some of the specifics of `allow` before we move on to some examples. We can `allow` different types of operations based on a condition that we create. This mostly just comes down to writing some kind of `if` test - you can see some

very basic examples of this above, but we will get into it in a little more detail in the next section.

The two generic types of operations are `read` and `write`, but these can be broken down into more specific operations:

read

- `get` - for single documents
- `list` - queries/collection read requests

write

- `create` - allows creation of new documents
- `update` - allows updates of existing documents
- `delete` - allows deleting documents

Depending on the circumstances, you may just want to use the blanket `read` and `write`, or you may want to get more specific as we did with the `comments` example above:

```
match /comments/{comment} {
  allow read;
  allow create: if true;
  allow update, delete: if false;
}
```

This would allow any kind of `read` on the comments, but instead of allowing all kinds of `write` operations it will specifically allow the creation of new documents. Any attempt to update existing documents, or delete them, would fail.

Creating and Testing Rules

In this section, we are going to focus on creating the rules for the application we are building. Rather than just jumping straight to the end result, we are going to expand upon our security rules step-by-step and learn along the way.

An important thing to keep in mind when writing these rules is that you can test them using the **Simulator**. You will notice this option in the bottom-left corner of your rules interface. Expanding this will provide you with an interface that you can use to run test read/write operations against your security rules:

The basic idea is that you would choose a **Simulation type** to reflect the kind of operation you are testing against, and then you would specify a **Location** which would be the specific document or collection you want to test your rules against. Where necessary, you can also toggle the **Authenticated** option to simulate an authenticated user attempting to access the document, and for **write** operations you can also build out a test document with specific fields.

For example, if I wanted to test read access to documents in my `messages` collection I would set the the `Simulation Type` to `get` and the `Location` field to `messages/{message}`. For the simulation, we need to point to a specific document (not the whole collection), so we can just use `{message}` to refer to a generic document in that collection.

At each step along the way in this lesson, I would recommend testing the rules we create with the simulator. Many of the rules we create initially will intentionally have security flaws, so see if you can simulate scenarios where the rules both pass and fail. You can start by testing our default "test mode" rules against the simulator.

Once you are done messing around with the simulator, let's write our first rule. Just copy and paste the following into your rules interface:

```
rules_version = '2';
service cloud.firestore {

  match /databases/{database}/documents {

    match /messages/{message} {
      allow read;
      allow write: if false;
    }

  }

}
```

Now, rather than creating a blanket rule for our entire database, we are specifically targeting the **messages** collection (even though this is our only collection in this example, so it doesn't make much of a difference). We've already talked about this particular rule in quite a lot of detail in the previous section, so we won't spend too much time on it here. The basic idea is that it will allow any reads, but it will deny all writes. See if you can verify this behaviour using the simulator by using both the `get` and `create` options.

Now let's move on to our next rule:

```
rules_version = '2';
service cloud.firestore {

  match /databases/{database}/documents {

    match /messages/{message} {
      allow read, write: if request.auth != null;
    }

  }

}
```

This is where things start to get a little more interesting, now we are specifically targeting users who are authenticated. We can access the `request` object inside of our rules in order to gain information about the specific user who is making the request. By checking `request.auth` we can tell if this is an authenticated user making the request. In this case, we are allowing any `read` or `write` as long as the user is authenticated.

This might sound good, but it still wouldn't be secure for most scenarios. This means that any user has complete access to the collection as long as they are authenticated. They could create documents, delete their own documents, delete others documents, edit others documents, and so on. This is getting closer to what we want, but it's not quite there yet. Let's try the next rule:

```
rules_version = '2';
service cloud.firestore {

  match /databases/{database}/documents {

    match /messages/{message} {
      allow read, create: if request.auth != null;
      allow update, delete: if false;
    }

  }

}
```

This is getting a lot better now. We have made our rules more granular to target more specific operations. We are still allowing all reads for any authenticated user, but we have split up our `write` into `create`, `update`, and `delete`. We want users to be able to add new chat messages, but we don't want them to be able to update or delete theirs or anybody elses.

With this set up, any authenticated user can create new documents and read all of the documents. Nobody is allowed to `update` or `delete` any documents. These rules are *pretty* good, but they still aren't entirely secure, and there are further improvements we can make. Let's try again:

```
rules_version = '2';
```

```
service cloud.firestore {

  match /databases/{database}/documents {

    match /messages/{message} {
      allow read: if isLoggedIn();
      allow create: if isLoggedIn() && isValidMessage();
      allow update, delete: if false;
    }

  }

  function isLoggedIn(){
    return request.auth != null;
  }

  function isValidMessage(){
    // uid of incoming doc should match the authenticated user
    return request.resource.data.uid == request.auth.uid;
  }

}
```

This brings us to our final rules, and we've done a couple of things here. First of all, we have split out our `allow` expressions into functions. Rather than repeating the same logic over and over in different rules, we can create functions that do the same thing for us. This allows us to reuse our logic, and can also make things a bit easier to read.

Keep in mind that there are some limitations to these functions, and they can't just be treated like normal JavaScript functions. For example, they are limited to a single `return` statement and you can't add additional logic like `if/else` statements or loops.

As well as refactoring our rules a little to utilise functions, we have also added some additional logic. Our `create` operations now not only check for an authenticated user, they also check for a "valid message".

Although this isn't something we are really concerning ourselves with in this application, we might want to verify that the `uid` associated with a message matches the `uid` of the user that is currently authenticated with Firebase. The user could mess with the data on the client-side before it is sent to the database, so to combat this we can run this little bit of logic:

```
return request.resource.data.uid == request.auth.uid;
```

As well as checking data from the currently authenticated user on `request.auth` we can also check the incoming document data by accessing `request.resource`. With this bit of logic, we are just checking that the `uid` on the incoming document matches the `uid` of the current authenticated user (and it should, as long as the user hasn't messed with the `uid`).

In order to test this rule, you would need to use the **Build document** option in the simulator and create a document with a `uid` field that matches the authenticated user. For the simulator, you can view the authentication payload by scrolling down to the bottom of

the options - it looks like this:

```
{
  "uid": "",
  "token": {
    "sub": "",
    "aud": "stencil-chat",
    "email": "",
    "email_verified": false,
    "phone_number": "",
    "name": "",
    "firebase": {
      "sign_in_provider": "google.com"
    }
  }
}
```

Since our authentication payload for the simulator just has a blank `uid` field, you can also just create a blank `uid` field in the document you are testing. Try building one document with a blank `uid` and a `create` operation should pass, then try building another document with a `uid` of `1` (or anything else) and you should see that it fails.

With our rules completed, all you need to do now is hit the **Publish** button at the top of the rules window and they will soon take effect. The simulator is a fantastic way to test your rules, but it's always worthwhile to test your rules using your real application as well. One

of the cool things about Firestore security rules is that the security implementation is quite readable/obvious. For example, just by looking at these rules:

```
match /messages/{message} {
  allow read: if isLoggedIn();
  allow create: if isLoggedIn() && isValidMessage();
  allow update, delete: if false;
}
```

We can get a pretty good idea of what is allowed since it almost translates directly to English:

- Allow reads if user is logged in
- Allow creates if user is logged in and the message is valid
- Never allow updates or deletes

I would like to reiterate again that there is still much more to learn in regard to both Firestore security rules, and security in general. Even the rules we have created here could be improved even further to improve security - perhaps we might want to add some kind of rate limiting to prevent people flooding the database with new messages, or perhaps we might want to enforce the structure of our documents. But this is enough for the example application we are building and it should give you a pretty decent basic understanding to start building off of.

Lesson 8: Styling & Animations

In this lesson, we're going to modify our templates a bit to add some classes and we will create some custom styles, as well as overwriting some of the app wide SASS variables. If you've completed some of the other applications then you'll probably know that there's usually not too much work to be done in these lessons, and this one isn't that different, but we will be doing something a little bit different by including some custom animations.

Animations, when done right, can do a lot to help make your application look and feel more high quality. When done poorly they can look tacky and can also cause a big performance hit.

Basic Styling

Let's start off by adding some basic styling throughout the application to make it look a little nicer. First, we're going to modify the **variables** file to make some application wide changes.

> Change the primary color in `src/theme/variables.scss` to reflect the following:

```
--ion-color-primary: #54a0ff;
```

We're just going to start by changing the primary colour for the application. We also want to define a few application wide styles in the global style file as well.

> Add the following styles to `src/global.scss`:

```
ion-title img {
  max-height: 39px;
  margin-top: 6px;
}

ion-header ion-toolbar {
  box-shadow: 0px -7px 11px 0px #0e103e;
}

ion-menu {
  ion-content {
    --ion-background-color: #3c8df0;
  }

  ion-list {
    --ion-item-background: transparent !important;
  }

  ion-item {
    color: #fff;
  }
}
```

We're modifying the position of the logo a bit and making the links inside of the menu have a transparent background. Next, we are going to modify our login page and home page to

include some custom styling.

> Modify `src/app/home/home.page.scss` to reflect the following:

```
ion-content {
  --ion-background-color: var(--ion-color-primary);
}

ion-list {
  padding: 0;
  background-color: #f3f3f3;
}

ion-textarea {
  font-size: 1.2em !important;
  --padding-top: 25px;
  --padding-start: 15px;
}

.chat-message {
  background-color: #fff;
  width: 100%;
  padding: 10px;
  border-radius: 10px;
  margin: 10px 0;

  p {
```

```
    margin: 5px 0;
  }
}
```

Nothing exciting going on here, we're just modifying the positioning of some of the layout elements so they sit more nicely, and adding a bit of extra styling to our messages and chat input area.

> Modify `src/app/login/login.page.scss` to reflect the following:

```
.container {
  background-color: var(--ion-color-primary);
  width: 100%;
  height: 100%;
  padding: 30px;
  text-align: center;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}

.logo {
  max-width: 250px;
  height: auto;
}
```

```
.login-description {
  line-height: 2em;
}

.login-description p {
  color: #fff;
}

.login-button {
  box-shadow: 0px 6px 20px 0px rgba(0, 0, 0, 0.18);
}
```

That's it for the basic styling, the pages in your application should now look something like this:

CamperCHAT
Camper Chat allows you to talk to other caravaners, RV'ers, road trippers and campers near you. Use Camper Chat to give and receive tips, ask for help with your flat tyre, or just have a friendly chat.
Connect with Facebook

CamperCHAT
Joshua Morony
We're staying for a few days or so
Kathy Sara
We just left there! The parks were all pretty full
Kathy Sara
but there were some good free spots you could check out 😄
Joshua Morony
Thanks!
type message...

We still have the most fun bit left to do though, and that's adding a couple of animations.

Creating Animations

One important thing to remember when animating elements is to use the **translate3d** property (even though we are only animating through 2D space). By using this property rather than animating the left property for example, the devices GPU (Graphics Processor Unit) is invoked. This is called a hardware accelerated animation and is a lot smoother. You may find that animations that run fine on your desktop browser don't work so well on mobile devices, so it’s usually important to use hardware acceleration (although relying on the GPU too much can also be disadvantageous, as it can lead to battery drain).

To create our animations we are going to define our own **keyframes**. Basically, this allows you to specify what certain properties should be at a specific stage during an animation. For example:

```
@-webkit-keyframes slideInSmooth {
  0% {
    -webkit-transform: translate3d(-100%, 0, 0);
  }
  100% {
    -webkit-transform: translate3d(0, 0, 0);
  }
}
```

What we are saying here is that at the start of the animation (0%) the element should be displaced 100% to the left (the three parameters in translate3d represent the x, y, and z axis), so it will be just off screen. Then at the end of the animation (100%), the element should be back to its normal starting position.

We could also define a similar animation like this:

```
@-webkit-keyframes slideInSmooth {
  0% {
    -webkit-transform: translate3d(-100%, 0, 0);
  }

  50% {
    -webkit-transform: translate3d(50%, 0, 0);
  }

  100% {
    -webkit-transform: translate3d(0, 0, 0);
  }
}
```

Now the element would start off to the left, then go to the right and then finally get back to its normal position. You can define as many of these intermediate steps as you like, the only important thing is that you always have a 0% and 100% (alternatively, you can use from and to) otherwise the animation will be invalid.

We can attach these animations to any element simply by creating a class that uses the animation and attaching the class to any element. We aren't going to be using these animations, but we will create some similar ones to animate in both the user's display picture and their message.

> Add the following styles to `src/app/home/home.page.scss`:

```
@keyframes animateInPrimary {
  0% {
    transform: translate3d(-100%, 0, 0);
  }

  100% {
    transform: translate3d(0, 0, 0);
  }
}

@keyframes animateInSecondary {
  0% {
    opacity: 0;
  }

  50% {
    opacity: 0;
  }

  100% {
    opacity: 1;
```

```
  }
}

.animate-in-primary {
  animation: animateInPrimary;
  animation: animateInPrimary;
  animation-duration: 750ms;
}

.animate-in-secondary {
  animation: animateInSecondary ease-in 1;
  animation: animateInSecondary ease-in 1;
  animation-duration: 750ms;
}
```

We've created two different animations here. The **animateInPrimary** animation will make the element start off outside of the left bounds of the screen, and will slide into its normal position evenly over the course of the animation. The **animateInSecondary** animation will start fading in from being fully transparent to full opaque halfway through the animation.

We assign these animations to two separate classes, both of which set an animation time of 750ms. Now to use these, all we have to do is apply the **animate-in-primary** and **animate-in-secondary** classes to an element.

> Modify the list in `src/app/home/home.page.html` to reflect the following:

```
<ion-list lines="none">
  <ion-item *ngFor="let message of messages">
    <ion-avatar
      [attr.slot]="message.displayName == user.displayName ?
'start' : 'end'"
      class="animate-in-primary"
    >
      <img [src]="message.displayPicture" />
    </ion-avatar>
    <div class="chat-message animate-in-secondary">
      <ion-note>{{ message.displayName }}</ion-note>
      <p>{{ message.message }}</p>
    </div>
  </ion-item>
</ion-list>
```

Notice that we've added the primary animation to the avatar, and the secondary animation to the message content area. If you load up the application now you should be able to see these elements animate in when the chat messages load in!

Conclusion

Congratulations on making it through the Camper Chat application. We've learned a lot through developing this application, but the main points are:

- Authentication with Firebase and Facebook
- Implementing different behaviour based on the platform
- Navigation/Route protection
- Using a Sliding Menu
- Using Firestore to store data
- Updating and displaying data in real time
- Integrating a 3rd party SDK

There's always room to take things further though, especially when you're trying to learn something. Following tutorials is great, but it's even better when you figure something out for yourself. Hopefully, you have enough background knowledge now to start trying to extend the functionality of the application by yourself, here are a few ideas to try out:

- Make the user's device vibrate when a message is received by using a native plugin **[EASY]**
- Modify the authentication process to display useful error messages to the user if the authentication fails **[MEDIUM]**
- Add "Channel Categories" that will allow the user to switch between different chat rooms, e.g: Vehicle Issues, General Chat, Camping **[VERY HARD]**

Remember, the Ionic documentation is your best friend when trying to figure things out. If

you want to tackle the last challenge above, you will certainly want to check out a lot more of the Firebase documentation as well.

What next?

You have a completed application now, but that's not the end of the story. You also need to get it running on a real device and submitted to app stores, which is no easy task. The final sections in this book will walk through how to take what you have done here, and get it onto the app stores so make sure to give that a read.

Testing & Debugging

When creating your application you are, without a doubt, going to make some mistakes and run into some errors. It's not always obvious where you've gone wrong either, so it's important to know how you can go about tracking down the problem.

This lesson is going to cover how best to debug Ionic applications, but it won't cover what debugging is in general or how to use debugging tools (e.g. viewing the source code, setting breakpoints, looking at network requests and so on). If you are not familiar with browser based Javascript debugging, then I highly recommend having a read through of this first. It's not important that you have a solid understanding of browser debugging principles, you just need to know enough to find your way around a little bit.

Understanding Errors

Until you have experience with dealing with errors, they are probably going to be confusing and not provide you with any obvious solution (although, sometimes error messages will provide specific instructions for fixing common errors).

In the beginning, the best approach is going to be to get a handle on figuring out what the important part of the error message is, and then Googling that to find out what to do. Eventually, you will become more familiar with various errors and have a better understanding of how your project needs to be pieced together, and you won't need to rely on Google as much.

The first place to look for errors should be in the command line interface when you `serve`

your application. Before the application can successfully run in the browser it needs to be compiled, and you may run into an issue at this stage. A typical CLI error might look like this:

```
ERROR in src/app/app.component.ts(25,25): error TS2551: Property
'hiden' does not exist on type 'SplashScreen'.
```

These errors are usually pretty straight-forward to interpret. We get the location and line number of the error, and most of the time it will be an error like this where we are trying to access some unknown property on an object. In this particular case, I have a typo because I am trying to access the `hiden()` method (which does not exist) instead of `hide()`.

If the application successfully compiles, then your application may run in the browser, but you can still face errors at runtime. At this point in time, you should open up the browser debugging window by right-clicking and choosing `Inspect Element`. There are a lot of debugging tools here, but for errors that break the execution of your application, you will mostly be looking at the `Console` tab.

In the `Console` you will see errors that might look something like this:

```
ERROR Error: StaticInjectorError(AppModule)[AppComponent ->
SplashScreen]:
  StaticInjectorError(Platform: core)[AppComponent ->
SplashScreen]:
```

```
    NullInjectorError: No provider for SplashScreen!
    at
NullInjector.push../node_modules/@angular/core/fesm5/core.js.NullIn
 (core.js:936)
    at resolveToken (core.js:1175)
    at tryResolveToken (core.js:1120)
    at
StaticInjector.push../node_modules/@angular/core/fesm5/core.js.Stat
 (core.js:1015)
    at resolveToken (core.js:1175)
    at tryResolveToken (core.js:1120)
    at
StaticInjector.push../node_modules/@angular/core/fesm5/core.js.Stat
 (core.js:1015)
    at resolveNgModuleDep (core.js:8065)
    at
NgModuleRef_.push../node_modules/@angular/core/fesm5/core.js.NgModu
 (core.js:8753)
    at resolveDep (core.js:9118)
```

This is an example of an error that is a bit harder to figure out. As a beginner, reading this likely won't make any sense at all. This is a rather common error though, and with a little experience you will realise that the important part of this error message is:

```
NullInjectorError: No provider for SplashScreen!
```

This means that we have forgotten to add `SplashScreen` as a provider in `app.module.ts` - this isn't immediately obvious. This error includes the "stack trace", which is basically the entire path the code has taken to get to that error. Most of this is entirely useless because it is just describing the internals of Angular which we aren't interested in (at least not for fixing our error).

Until you become more familiar with different errors, it's important to be able to work out what the important part of the error message is, and then Google that for assistance. A good way to spot the useful part of the error is to look for:

- Parts that are descriptive not just identifying a location (i.e. ignore all the `at at at at at` parts)
- Errors that refer to files that you recognise/have modified

Dumping that entire error message into Google likely won't yield useful results, but if you just Google:

```
NullInjectorError: No provider for SplashScreen!
```

You will likely find a lot of people who have run into the same problem.

Browser Debugging

Your desktop browser will usually be your first point of call when developing your application. It's great to be able to debug directly through the browser because, especially

with live reload, you can quickly see the impact code changes have and iterate really quickly.

But it's important to keep in mind that **it is not representative of how the application will run on an actual device**. For the most part, how it behaves in the browser will be the same as the way it works on a real device, but there can be differences and in the case of Capacitor/Cordova plugins, they won't always work when testing through a desktop browser.

A good approach is to do the majority of your testing in the browser, and once you're getting close to being happy with how everything is working switch to testing on a real device to make sure everything still works correctly.

iOS Debugging

In order to debug your application on an iOS device, you will need to first follow some of the steps in the Building for iOS and Distributing to the Apple App Store section to actually get your application running on the device.

Once you have the application running on your device, there are two primary ways to see what is happening:

1. The debug output displayed in Xcode
2. Remotely using browser debugging tools

You can actually debug your application just like you would when you are running it on your computer. Using the Xcode debug output is a good way to spot some errors, but for more in-depth debugging it is useful to use the browser debugging tools. In order to do

this, all you need to do is first enable debugging by enabling the following settings on your iOS device:

- Go to `Settings > Safari > Advanced` and turn `Web Inspector` and `Javascript` on

You will also need to enable the `Develop` menu in Safari (on your computer) by going to:

```
Safari > Preferences > Advanced > Show Develop menu in menu bar
```

Now all you need to do is plug in your iOS device via USB, run the application through Xcode on your device. Once the application is running on your device, you can debug it using the Safari Dev Tools on your computer. Simply open up **Safari** and then go to:

`Develop > iPhone > index.html`

Once you open up **index.html** in Safari you should see a debugging screen like this:

Debugging this way is great to use throughout development, but you should also always

make sure to test the final build of your application using Test Flight before submitting it to the App Store.

Android Debugging

Debugging on Android is similar to iOS except that you will use Chrome instead. You will need to follow the steps in Building for Android and Distributing to Google Play in order to get the application running on your Android device.

You will need to have USB debugging enabled on your device for this to work. Enabling USB debugging is different for different devices, so just Google "[YOUR DEVICE] enable usb debugging". On your desktop browser (in Chrome) you will now be able to go to the following address:

```
chrome://inspect/#devices
```

and then click 'Inspect' on the device you want to debug on, and you will see the debugging tools:

You will now be able to use the debugging tools as you normally would on your computer.

Startup Errors

It is important to check for errors that occur right when the application first boots. These are quite common and can prevent your application from displaying anything at all. Since the remote debuggers aren't attached to the application right away (you first need to launch the application, and then bring up the debugger) it can miss startup errors. If you are running into issues and you don't seem to have any errors, make sure to check the logs in Xcode/Android Studio as well. You can also hit the refresh button through the debugging tools to reload your application with the debugger already active.

Creating a Production Build

By default, when running your application Ionic will use a development build. This makes things easier to debug, but it's not the final minified format that is used for production builds, so your application may not be as fast as it can be. To run a production build, just add the `--prod` flag to the command, i.e:

```
ionic build --prod
```

If you are testing your build on an actual device, make sure to create a production build and copy it over to the native project with Capacitor

```
ionic cap sync --prod
```

Summary

Debugging applications can be a difficult and frustrating task, especially when debugging on a real device, but over time you tend to develop a good sense of where things may have gone wrong and the process becomes a lot easier. If you're stuck on a particular error, you can always head over to the Ionic Forum for help, just make sure to provide as much detail about the error and what you've tried as you can.

Building & Submitting

Preparing Assets

Before we get into creating the final builds of our application, we will need to set up some graphical assets for the splash screens and icons. If you are just building for testing purposes then you don't really need to worry about this (you can just use the defaults), but if you are building the final production version of your application, then you will want to use your own custom splash screens and icons.

Your custom icon will be displayed in various places, but mostly it will be used to represent your application on the user's home screen or application menu. The splash screen is the image that displays whilst your application is loading (i.e. as soon as you tap to open the application, this image will be displayed briefly). Depending on the context, splash screens are also referred to as launch images.

Generating icons and splash screen images end up being a much more difficult task than you might anticipate before doing it for the first time. The difficulty stems from the fact that there are a bunch of different devices that your application could be running on, a lot of which will have different resolutions/pixel density. Both iOS and Android applications have systems in place to pull in images of the correct resolution for the particular device the application is running on, but that means that we need to supply a lot of different resolution icons and splash screens.

If you have previously used Cordova to build applications, you may be used to splash

screens and icons being defined in the **config.xml** file. In Capacitor projects, we can just manage the splash screens and icons directly in the native project like any normal native application developer would. Capacitor already has a default set of icons and splash screens provided, but in this lesson, we are going to look at how to replace those with our own images.

I will assume at this point that you already have Capacitor set up and have your desired platforms adds, e.g:

```
ionic cap add ios
```

```
ionic cap add android
```

If you haven't, you should add your desired native platforms now (if you are unsure about how to get Capacitor set up, you can revisit the **Getting Ready** lesson of any of the application walkthroughs).

Generating the Icon and Splash Screen Assets

The first step is to prepare the assets that we will be using for our icons and splash screens. As I mentioned, there are a lot of different sizes/resolutions we need to cater for, so it is often easier to use some kind of tool to help us create these assets.

If you are interested in seeing the variously sized assets we require for **iOS**, you can check

out the documentation: Icons | Launch Screens.

It is going to be a little easier for **Android** because we can use Android Studio to automatically generate and set up the icons for us, so we don't need an external tool to help us with this. You may still want to use a tool to generate splash screens for Android, though.

There are quite a few tools out there to help with this task. Depending on your workflow and preferences you will probably prefer using different tools and methods, here are some options:

- Just create every required icon and splash screen size manually
- MakeAppIcon - great for creating iOS icons, but doesn't create splash screens or Android assets
- ImageGorilla - creates icons and splash screens for iOS and Android (but some formats are missing)
- SplashScreen Pro - this is a paid tool but integrates directly with Photoshop or Sketch

However, one particularly good option is cordova-res which has been published by the Ionic team. It was originally developed for use with Cordova, but it will also work with Capacitor and other native runtimes. All we need to do is install it globally with the following command:

```
npm install -g cordova-res
```

Before we can use `cordova-res` to generate our icons and splash screens, we need a base asset for it to work from. In the root folder of our project, we should create a **resources** folder, and inside of that we should create a **icon.png** and **splash.png** file, i.e:

- resources
  - icon.png (at least 1024x1024px)
  - splash.png (at least 2732x2732px)

The tool will automatically resize/crop the resources as necessary - this is why they should be large. Icons are simple enough since they are square, but to ensure that you don't get undesirable cropping for splash screens, make sure the most important elements of your splash screen are in the middle. Any elements that are closer to the edges might get cropped for different devices. You can always just run the tool and check the assets that are created to see if you are happy with them.

NOTE: For Android, you can also use the 9-Patch File approach to create adaptable splash screens. This method is somewhat more complex, but gives you greater control over how your splash screen will be adapted.

When using `cordova-res` with Capacitor, we should supply some additional configuration options. To generate the resources for iOS we should run:

```
cordova-res ios --skip-config --copy
```

and for Android:

```
cordova-res android --skip-config --copy
```

These additional options will prevent `cordova-res` from writing to the `config.xml` file that is only required for Cordova, and also copy the resources over into the native project.

If the command has run successfully you should see a message similar to this for iOS (although the exact number of resources might change over time):

```
Generated 47 resources for iOS
Copied 21 resource items to iOS
```

But on Android, you might receive a warning like this:

```
Generated 18 resources for Android
WARN:    Error occurred while copying
resources/android/icon/mdpi-foreground.png
WARN:    Error occurred while copying
resources/android/icon/hdpi-foreground.png
WARN:    Error occurred while copying
resources/android/icon/xhdpi-foreground.png
WARN:    Error occurred while copying
resources/android/icon/xxhdpi-foreground.png
WARN:    Error occurred while copying
```

```
resources/android/icon/xxxhdpi-foreground.png
Copied 26 resource items to Android
```

This is because the tool will also try to copy over Adaptive Icons for Android. To support Adaptive Icons, we need to add the following additional resources:

- resources/android/icon-foreground.png (432x432px)
- resources/android/icon-background.png (432x432px)

The foreground should be something like your applications logo on a transparent background, and the `icon-background` can be whatever you want. If you just want to use a simple plain colour background, you don't need to worry about creating an image file for that. You can just supply the `icon-foreground` and when running the generate command supply the hex colour value for the background, e.g:

```
cordova-res android --skip-config --copy --icon-background-source
'#E74C3C'
```

If this is successful, you should now see a message like this instead:

```
Generated 24 resources for Android
Copied 26 resource items to Android
```

Verifying Resources

To check that your resources are set up properly you can view them in the native project. To view your Android icons you should open up your project in Android Studio:

```
ionic cap open android
```

...and then go to **app > res > mipmap**:

This is where you will find all of your icons. In the **values** folder you will also find a **ic_launcher_background.xml** file which contains values for your adaptive icons. At the time of writing this, the value I supplied to the generate command for the background wasn't used, so I had to manually update:

```
<color name="ic_launcher_background">#FFFFFF</color>
```

to:

```
<color name="ic_launcher_background">#E74C3C</color>
```

To find your splash screens you can go to **app > res > drawable > splash**.

To view the iOS icons and splash screens you should open up your project in Xcode:

```
ionic cap open ios
```

If you then navigate to **App > App > Assets.xcassets** you will be able to view both the icons and the splash screens in the same location:

You might notice that a "universal" splash screen is being used. The benefit of using a universal image for the launch image is that you only need to create a single design, and that can be used across all devices. The downside to this is that the image will be cropped to fit devices, so it's important that the important parts of your design are centered. If you would prefer, you can create specific designs for each of the available resolutions to make better use of the available space.

Summary

Creating icons and splash screens is usually tedious work that is done right at the end of your project when you're eager to get everything wrapped up and submitted. However, you should take some time with this to make sure you get everything right. You don't want an ugly low-resolution icon, or the incorrect icon, to be the first thing a new user sees when using your application.

With the help of a variety of tools that are available, it's not too bad!

Building for iOS and Distributing to the Apple App Store

In this lesson, we are going to walk through how to use Capacitor to deploy iOS applications to a device, and to the Apple App Store. We will be covering how to get a Capacitor application running on an iOS device for development, all the way through to creating a build for distribution on the App Store.

Keep in mind that although we are using Ionic/Angular in this book, you can use Capacitor to build and deploy any web based code. To recap, Capacitor performs two main roles:

1. Wrapping a web-based application in a native shell
2. Facilitating communication between the web application and the Native APIs of the device the application is running on.

All you need to build your application with Capacitor is some web-based code to point it at, and so that doesn't *need* to be an Ionic application.

Do I Need a Mac to Deploy to iOS?

Short answer: **Yes**. To follow the steps in this lesson you will need a Mac. **Xcode** is required as it provides the necessary SDKs for building iOS applications, and Xcode will only run on macOS. If you do not have a Mac, some options available to you are:

- Using Appflow to handle creating the iOS build for you.
- Using a service like macincloud to access a Mac machine remotely
- Borrow a friends machine - if you are using something like Ionic, you can do most of

your development in the browser. Although it helps to have access to native builds, you may be able to get away with just borrowing somebody else's computer to perform the build step.

If you are able to get your hands on a computer with macOS that you can use on a frequent basis, it does help a great deal. You can do *most* of the work for iOS on a non-macOS machine, just not the final build and submission step (which is obviously quite frustrating).

1. Sign up for the Apple Developer Program

In order to create iOS applications, you will need to enroll in the Apple Developer Program which has an annual fee of $99 for individuals and $299 for enterprises at the time of writing this.

2. Install Xcode

In order to run applications on your iOS device, and submit them to the App Store, you will need Xcode. You can download this from the Apple App Store.

As well as installing Xcode itself, you will also need to install Cocoapods and the Xcode CLI tools. You can do that by running the following commands:

```
sudo gem install cocoapods
```

```
xcode-select --install
```

Once you have installed Xcode, make sure to open it at least once so that it can install the necessary components.

3. Install the LTS Version of NodeJS

You will need to have Node installed on your machine in order to use Capacitor. I would recommend downloading the LTS version of NodeJS from the NodeJS website.

It is also a good idea to install the n package (unfortunately, n is not available for Windows - you do not need it, it's just a helpful tool to have). This is a version manager for Node, and allows you to easily switch to whatever version of Node you want to use. Capacitor requires at least Node version 8.6.0 for example, but other tools you use may require different versions. Using n allows you to run a one-line command to switch between versions. To install n just run the following command (after installing NodeJS):

```
npm install -g n
```

Then you can switch to whatever Node version you like using the following command format:

```
n 14.15.1
```

As a default, you should use whatever the current **LTS** version is (as listed on the website).

4. Create Your Web Application

At this stage, you will need your web application that you want to deploy to iOS. Presumably, at this stage, you already have an Ionic application that you want to build, and it is important that you build your application using the `prod` flag:

```
ionic build --prod
```

This creates an optimised production build of your application, and this is what Capacitor will pull into the native project. When running the sync command for Capacitor, it is also important to supply the `--prod` flag:

```
ionic cap sync --prod
```

Otherwise, Capacitor will just trigger a standard development build of your Ionic project before copying the resources over.

5. Install Capacitor

You should also have Capacitor installed at this stage, and you should have added the native platforms that you are targeting. If you have not, go back and follow the steps in one

of the **Getting Ready** lessons for the example applications.

Make note of the identifier/bundle ID that Capacitor has used for your project in **capacitor.config.json**. You will need to use this later to create an identifier for your application with Apple. If you want to change the Bundle ID after already having added the `ios` platform, you can just change it in the **capacitor.config.json** file and also under **General > Bundle Identifier** inside of your Xcode project (just click on **App** in the project explorer to bring up the tabs, the first of which will be **General**).

When you are ready to build, you should make sure that all of your code and plugins are synced with Capacitor by running the following command:

```
ionic cap sync --prod
```

You do not need to run the `sync` command each time you make changes to your project. If you are only making changes to the web code of your application, you can run the following command instead:

```
ionic cap copy
```

which will update your native project with your web code. To open your project in Xcode, you can run the following command:

```
ionic cap open ios
```

From this point onwards, we are basically just dealing with a normal native project. We just have our web code bundled in there.

6. Running the Application on an iOS Device

In the image above, you will see a little play icon near the top left. This is what you will use to run your application. By default it will be set to 'Generic iOS Device', but if you have your own device plugged in via USB you will be able to set it to run on that. You can also choose a simulator to run the application in if you like.

Let's try and run the application now (make sure to attach and select your iOS device if you have one you want to use). When you are ready, click the 'Play' button.

At this point, you may get an error that looks like this:

```
Signing for "App" requires a development team. Select a
development team in the project...
```

```
Code signing is required for product type 'Application'
```

Apple requires that applications are "signed" in order to run on a device (and to be distributed on the App Store). This is basically a way to prove ownership of that application and to associate it with a particular Apple ID. In order to fix this error, you will need to set up signing for the application.

7. Sign the Application

To set up signing for the application, click **App** in the file explorer for the project in Xcode:

and then choose the **Signing & Capabilities** tab:

Expand the **Team** dropdown and click the **Add account** button to add a development team. All you need to do here is sign in with your account that you purchased the [Apple Developer Program]() for.

Once you have signed in, make sure to change the **Team** to your account:

You will likely still get an error at this point that looks like this:

```
Failed to create provisioning profile.
There are no devices registered in your account on the developer
website. Plug in and select a device to have Xcode register it.
```

You will need to have at least one device registered with Apple in order to create a provisioning profile. To do this, plug in the iOS device you want to register and then select it in Xcode (just click the device dropdown to the right of the 'Play' button and select the device you plugged in). You should then see an error message like this:

```
Device "Your iPhone" isn't registered in your developer account.
The device must be registered in order to be included in a
provisioning profile.
```

Click the **Register Device** button and Xcode will handle this automatically for you. Now you should be able to hit the 'Play' button again, and this time the application should be installed on your device.

Anytime you want to test changes you can just rerun `ionic cap sync` and then hit the 'Play' button again. Alternatively, you can use Live Reload if you need to frequently test changes on a native device. With live reload, you get a similar experience to what you get in the browser - just save changes to your code and it will automatically appear in the native device. It is still a good idea to test the application by running it normally without live reload at some point.

Distributing on the App Store

We have the application running on a device now, but there are a few more steps before we can distribute our application to everybody on the app store.

Before distributing your application, make sure you are submitting a production build of your application, and have run `ionic cap sync --prod` to copy that build to your native project.

NOTE: I will just be walking through the basic steps here. Submitting to the App Store is a lengthy process that involves setting up your splash images appropriately, supplying screenshots for the App Store, providing descriptions of the application, creating icons for the application and so on. If you are submitting your production application to the app store, make sure that you also complete all of these steps.

1. Create an Identifier

We will require an "Identifier" for our application, which is basically just a name/id associated with the application through the developer portal. You should first log in to your Developer Account and then go to:

- Certificates, IDs & Profiles

In this section, we are going to create an **Identifier** for our application. To do that, follow these steps:

- Select **Identifiers**
- Next to the Identifiers title hit the '+' button
- On the following screen you should choose **App IDs** and then click **Continue**
- Select **App** as the type

You should now see a screen like this:

Fill out the details on this screen. When entering an explicit App ID, make sure that it matches the ID that you are using for your native projects (e.g. `com.example.yourapp`). Make sure that you also enable any capabilities that your app requires (if any). When you are ready, click **Continue**.

Review the information on the next screen, and hit **Register**.

NOTE: Due to the automatic provisioning in XCode, you might get an error saying that the Bundle ID you have specified can't be used. If that happens, just change the Bundle ID to something else - it doesn't really matter what the Bundle ID, it's just an identifier, just use

something that makes sense to you. If you do change the Bundle ID **you will need to update the Bundle Identifier in Xcode as well**.

2. Create an Application in App Store Connect

In order to submit an application, we need to create a record for it in App Store Connect. This is where we will be able to supply the screenshots, descriptions, pricing, and so on. Basically, all of the "meta" information for the application. It is also the interface you use for submitting your application for review, and for managing updates.

You should now open App Store Connect. From here, you will see an interface that looks like this:

On this screen, click on **My Apps**, and then click the '+' button at the top left on the new page and choose **New App**. Fill out the information in the box that pops up.

You will need to make sure that you use the identifier that you just created. You can use whatever SKU you like (I just used the date for this example). Once you complete this step, you will be taken to the page that will allow you to specify all of the "meta" information for

your application. Feel free to complete this now, or come back later and do it. A lot of the information will need to be filled out in order to submit, for a list of what exactly is required I would recommend checking out App Store & Google Play: Submission Checklist.

3. Sign and Upload the Application to App Store Connect

Let's head back to Xcode now. You should change the target device near the 'Play' button back to **Generic iOS Device**. Once you have done this, go to `Product > Archive`, and you should see a screen with your application should pop up.

The process of archiving is basically converting your application into the format required to upload it to the App Store. You will still have a chance to test your application through **Test Flight** once you have uploaded it to App Store Connect (and you should definitely do this to make sure everything is OK before submitting for review). Click **Distribute App** and choose **App Store Connect** as the method of distribution. Choose the **Upload** option and click **Next**. Keep all the default distribution options (if you want) and click **Next**.

Xcode will tell you that this app needs to be resigned for App Store distribution as it requires a distribution certificate, not a development certificate (which we can use to run the application on our own device). Unless you have reason to do otherwise, just use the **Automatically manage signing** option.

If you do not already have a distribution certificate created, you will see something like this:

Just select, **Generate an Apple Distribution Certificate** and then choose **Next**.

IMPORTANT: The next screen will inform you that the private key for your certificate is stored in the keychain locally on the computer. If something happens to your computer, or you forget your password, it means you will lose access to this. You should export this signing certificate and keep it somewhere safe. It is critically important that you do not lose this or the password used. Once you have done that click **Next**.

The next screen will allow you to review your information:

If you are happy, just hit **Upload**. Once the application has successfully uploaded you should see an upload successful screen:

Woohoo!, almost there. Once the app is uploaded, you will be able to associate it with the app record you created in [App Store Connect](). To do that, open your app in App Storee Connect and click on the current version of the application on the left of the screen (e.g. **1.0 Prepare for Submission**).

Go to the **Build** section (you will need to scroll down a little). If your upload has finished processing, you should see a '+' icon next to 'Build'.

NOTE: You may not see any builds available right away, as it takes some time to process. You should receive an email from Apple when your build is ready to be attached. When you do receive the email from Apple you may get a warning that your application is: **Missing Push Notification Entitlement**. This is fine to ignore. Capacitor includes code for managing push notifications by default, which causes Apple's verification to think that you might have forgot to enable push notifications. If you are not using push notifications, you

don't need to add the push notification entitlement. If you really want to get rid of this warning, you can see how in the Capacitor documentation.

Click the add icon in the build section, select the build you just uploaded, and then click **Done**. If you have not already, you should configure the rest of the app information in App Store Connect, which will vary depending on the type of app you are submitting.

Once you have absolutely completed everything, and you have tested your application to ensure everything is working as you expect, hit the **Submit for Review** button at the top right:

Now you can cross your fingers and hope that your application is approved! You can use App Store Connect to check on the current status of your application (e.g. to see if it has been successfully submitted, rejected, or something else).

Summary

It is quite a lengthy and complicated process to submit to the Apple App Store, but fortunately, Xcode does a great deal to simplify this with its automatic signing. A key point to using Capacitor is that you are mostly just working with a standard native iOS project, so most of what we have done here is the same as what any iOS developer would do to submit an application. There isn't really any specific configuration required for a Capacitor project.

Building for Android and Distributing to Google Play

In the last lesson, we walked through how to build Ionic/Angular applications for iOS and distribute them on the app store. In this lesson, we are going to be doing the exact same thing, except we will be deploying an Android application to Google Play.

Keep in mind that although we are using Ionic/Angular in this book, you can use Capacitor to build and deploy any web based code. To recap, Capacitor performs two main roles:

1. Wrapping a web-based application in a native shell
2. Facilitating communication between the web application and the Native APIs of the device the application is running on.

All you need to build your application with Capacitor is some web-based code to point it at, and so that doesn't *need* to be an Ionic application.

1. Register for Google Play

Unlike iOS development, you can build and submit Android applications no matter what operating system you are using. However, you will still need to register for a developer account and pay a $25 USD fee (this is just a one-time fee).

If you plan on submitting your application to Google Play, you should register an account now as we will need to use it later in the lesson. You can still build Android applications and distribute them through other means without registering for Google Play.

2. Install Android Studio

Much like we used Xcode in the previous lesson to build our iOS application, we will be using Android Studio to build our Android application. If you do not already have Android Studio, you should install it by visiting this website.

Once it is installed, you should open it. The first time you open Android Studio you will be taken through a setup wizard:

Just select the **Standard** options. When asked to choose between the *light* side and the **dark** side, choose wisely:

The wizard will then handle installing all of the dependencies you will need, which includes things like the SDK (Software Development Kit) for development, an emulator, and build tools.

Hit **Finish**. If you need to install or update any components in the future, you can do so using the **SDK Manager** that comes with Android Studio by going to **Tools > SDK Manager**. For Capacitor, at the time of writing this, we will need the Android SDK Tools (26.0.1 or greater) and a version of the Android SDK Platforms for API 21 or greater. If you are having any difficulty with this step, there is more information available here.

3. Install the LTS Version of NodeJS

You will need to have Node installed on your machine in order to use Capacitor. I would recommend downloading the LTS version of NodeJS from the NodeJS website.

It is also a good idea to install the n package (unfortunately, `n` is not available for Windows - you do not need it, it's just a helpful tool to have). This is a version manager for Node, and allows you to easily switch to whatever version of Node you want to use. Capacitor requires at least Node version 8.6.0 for example, but other tools you use may require different versions. Using `n` allows you to run a one-line command to switch between versions. To install `n` just run the following command (after installing NodeJS):

```
npm install -g n
```

Then you can switch to whatever Node version you like using the following command format:

```
n 14.15.1
```

As a default, you should use whatever the current **LTS** version is (as listed on the website).

4. Create Your Web Application

At this stage, you will need your web application that you want to deploy to iOS. Presumably, at this stage, you already have an Ionic application that you want to build, and it is important that you build your application using the `prod` flag:

```
ionic build --prod
```

This creates an optimised production build of your application, and this is what Capacitor will pull into the native project. When running the sync command for Capacitor, it is also important to supply the `--prod` flag:

```
ionic cap sync --prod
```

Otherwise, Capacitor will just trigger a standard development build of your Ionic project before copying the resources over.

5. Install Capacitor

You should also have Capacitor installed at this stage, and you should have added the native platforms that you are targeting. If you have not, go back and follow the steps in one of the **Getting Ready** lessons for the example applications.

Make note of the identifier/bundle ID that Capacitor has used for your project in **capacitor.config.json**. You will need to use this later to create an identifier for your application with Google. If you want to change the Bundle ID after already having added the `android` platform, the easiest way to do it is by changing it in **capacitor.config.json** and then deleting the `android` folder and then running `ionic cap add android` again. Keep in mind that if you do this any custom configuration you might have added to your Android project will be lost.

When you are ready to build, you should make sure that all of your code and plugins are synced with Capacitor by running the following command:

```
ionic cap sync --prod
```

You do not need to run the `sync` command each time you make changes to your project. If you are only making changes to the web code of your application, you can run the following command instead:

```
ionic cap copy
```

which will update your native project with your web code. To open your project in Android Studio, you can run the following command:

```
ionic cap open android
```

This will open your native Android project in Android Studio. The first time you open the project, it will perform a **gradle sync** which may take a little while. This will install all of the dependencies required for your project. Eventually, you should see that the sync has successfully completed:

Just like with the iOS lesson, we are now dealing with a standard native Android project. The steps from here on are the same as what you would do for building and submitting any native project.

6. Run on an Android Device

Now let's talk about how to run our Android application on an Android device. The Android Studio interface is also very similar to Xcode in terms of running an application. All you have to do is hit the **play** button on the top-right hand side of the screen.

If you have an Android device plugged into your computer, you will be asked to authorize it, and then you will be able to deploy to that device. If you prefer, you can hit the **Create New Virtual Device** button to run your application on an emulator instead.

From this point on, the workflow for deploying new builds that you want to test is:

1. Make whatever changes you want
2. Run `ionic cap copy` (if only web code has changed) or `ionic cap sync` (if you've made changes to plugins)
3. Run `ionic cap open android`

and then run as normal again through Android Studio. Alternatively, you can use Live Reload if you need to frequently test changes on a native device. With live reload, you get a similar experience to what you get in the browser - just save changes to your code and it will automatically appear in the native device. It is still a good idea to test the application by running it normally without live reload at some point.

IMPORTANT: When you are ready to submit your application, make sure to copy over a production build with `ionic cap sync --prod`.

7. Build a Signed App Bundle

Once you are ready to submit your application, you will need to build a **signed App Bundle** with Android Studio. An App Bundle is just the file that is the packaged up version of your application, and then that needs to be signed with a private key (keystore). This is essentially just a way of proving who the application belongs to, and who has the authority to update it.

Just like with iOS applications, there are quite a few steps involved with publishing your application, so I would recommend reading through the documentation available on the Android website.

We will just be walking through the basics steps of signing and uploading an App Bundle to Google Play. As I mentioned in the iOS lesson, make sure that the application you are distributing is a production build of your application (i.e. a minimised/optimised one built with the `--prod` flag).

In order to build your signed App Bundle, you should first go to **Build > Generate Signed Bundle / APK** in the menu at the top of Android Studio, choose **Android App Bundle**, and then click **Next**.

On this screen, you should supply your private key (keystore file) for this project, or if you do not yet have a keystore file you should select **Create New**. If you are not familiar with this interface for creating a keystore, I would highly recommend following the instructions here: Generate an upload key and keystore.

It is critically important to keep this keystore file safe, as you will need it to update your application on Google Play. If you lose it, it will be impossible to submit a new version of your application. When choosing a place to store your keystore file, it is a good idea **not** to include it in the files that will be uploaded somewhere like GitHub. You should also protect this file by choosing a strong password (but make sure not to lose it!).

Fill out the details in this prompt, and remember to make note of the password you use and the alias (you should use a separate password for the keystore and the key). Also make sure to tick the **Export encrypted key for enrolling published apps in Google Play App Signing** and export the encrypted key somewhere safe on your computer. When you are done, click **OK** and then **Next**.

Make sure to choose the 'release' build type, and then click **Finish**:

Once you complete this step, the application will begin building. Once it has finished you will be able to find your signed App Bundle at:

```
yourproject/android/app/release/app-release.aab
```

This is the file that you will need to upload to Google Play

7.1 Creating a Key Hash (Optional)

In some cases, you may be required to provide a key hash to identify your application. If you are integrating your application with the Facebook API, for example, you will need to supply a "key hash" to Facebook.

To create this key hash, you will need to run the following command:

```
keytool -exportcert -alias alias_name -keystore my-release-key.keystore | openssl sha1 -binary | openssl base64
```

Make sure to replace **alias_name** with your own keystore files alias name and **my-release-key.keystore** with the path to your keystore file. Once you do this, the key hash will be output to the terminal.

If you would like to generate a hash for the default debug keystore that Android Studio uses, you can run the following command:

```
keytool -exportcert -alias androiddebugkey -keystore ~/.android/debug.keystore | openssl sha1 -binary | openssl base64
```

You will need to enter the password: `android`. You can use this throughout development if you wish.

8. Upload to Google Play

The final step in this process is to upload that signed App Bundle file to Google Play. To begin, you should go to the Google Play Developer Console. Once you have logged in, click **Create app** and fill out the form.

At some point, you will need to fill in all of the details and promotional assets for the application. I'd recommend spending some time looking through all of the available

options, and reading advice from Google, to make sure that you have completed all of this information properly. Again, we are just focusing on uploading the App Bundle in this lesson.

In the dashboard, after you create your application, you will be guided through various steps - starting with supplying some information about your application. Complete these "tasks" in order to progress (this might take a little while!).

Once you have completed these tasks you will be able to go to the **Production** tab in the left hand menu to create a new release. Just click on the **Create new release** button.

Next, you should upload your signed App Bundle (located at `yourproject/android/app/release/app-release.aab`) to the 'App bundles and APKs' section, and fill out the **Release details** section.

Hit **Save** and then **Review release**. You will now be able to review any issues with your application, and once everything is good to go you will be able to select **Start Rollout to Production**. Remember, you will need to have completed all of the information for your application in Google Play before doing this step.

As long as you have done everything correctly, your application should be available on Google Play soon!

Summary

Just like with the iOS example, submitting an application to Google Play using Capacitor is no different than submitting any other native application once you've got the project in Android Studio. Although it isn't the easiest process in the world, being able to use tools

like Android Studio helps a great deal in simplifying the process.

Building for iOS and Android Using Appflow

Although the set of web components provided by Ionic are completely free to use, the Ionic team also do offer some optional paid services. One of these paid services is Appflow. Appflow is a platform to help manage the building and deployment of your Ionic applications. The features of Appflow include:

- A build service (for compiling your applications in the cloud, instead of handling it yourself on your local machine)
- A deploy service which will allow you to update your applications live on the app stores without requiring a new native binary submission
- A continuous integration service which will allow for automated builds/deploys

Although there are multiple powerful features provided by Appflow, the focus of this lesson is to quickly touch on the usefulness of Ionic Appflow's Package service.

Package will handle building your applications for you in the cloud, so you don't need to worry about dealing with the native SDKs on your computer. Although you may wish to use Appflow even if you do have a Mac, this feature is especially useful for people who want to develop iOS applications, but who do not have access to a macOS machine. The macOS operating system is required in order to compile native iOS applications, and so the Package service is a way to get around that requirement.

Ionic already have great documentation for setting up Appflow in your application, as well as using it to create native iOS or Android builds, so I won't rehash that here. To get started, just follow the instructions in the Quickstart tutorial.

In the past, it has been common even for these build services to still require a Mac in the end. The build service would generate the iOS **.ipa** package for you, but you still needed to use Mac hardware to upload that **.ipa** package to the app store. This is not the case with Appflow. Their **1-Click Publish to App Stores** feature allows you to submit your final built file directly from Appflow to the app stores.

Publishing as a PWA (Progressive Web Application) with Firebase

Progressive Web Applications (or PWAs) have had a big increase in popularity recently, and browsers (specifically iOS/Safari) have increased support for them which has made it a whole lot more viable to build a PWA.

In short, Progressive Web Apps are all about giving applications made available directly through the web (this means you just go to the URL through a web browser and you are instantly in the app) a more native-like experience. This means at a minimum that they should work offline, and can be installed onto a user's device (if a user "installs" a PWA, they can then access it directly from their home screen in the future rather than having to open a web browser to access it). Some of the key benefits of distributing your applications as a PWA are:

- Fast load times due to caching
- Offline capability
- Ease of distribution/low barrier to use
- Instant updates (no application review process)
- No "walled garden" app stores (i.e. Apple can't reject, remove, or ban your application)

The biggest downside is that you don't have direct access to all of the native functionality of the device. Capacitor helps a great deal with this because it provides a consistent API to use across iOS/Android/PWA, and many features like the Camera and Geolocation *are* still available when building a PWA. However, if you need access to native functionality that is

not available through a Web API, then the PWA version of your application will not be able to access it. This website is good for visualising what the web can and can not do today. In future, the list of things that the web can't do will likely get smaller.

This is a separate concept to what is typically referred to as a *hybrid* application, which you can also build with Ionic. A hybrid application consists of a web view embedded into an actual native application, the resulting package is no different from any other native application and as such, it is generally distributed through app stores. Hybrid applications are what we have discussed in the previous lessons, where we package the Ionic application for distribution on iOS/Android. These hybrid applications **do have access to the native functionality of the device**, because they *are* native applications.

That is the cool thing about using Ionic and Capacitor. We can build a single application, using a consistent API for accessing native functionality and distribute it as a hybrid application for iOS/Android and as a PWA for the web. We just need to take a couple of extra steps to enable PWA support by adding a "service worker" and configuring a few things.

In this lesson, we are going to be walking through the basic steps for enabling PWA support for your application. PWAs really deserves an entire book of their own, and there's more to Progressive Web Apps than just enabling a service worker for caching and offline support, but we get *a lot* of value just with those simple steps.

What is a Service Worker?

A service worker allows us to serve locally cached content to the user, rather than having to load that resource from a server. This means that we can have an application hosted on the web that will still continue to function even when the user accesses that application

offline. The basic idea behind a service worker is that it will intercept all network requests, and then it can decide how to handle those requests. If your application makes a request to `https://www.somewhere.com/somefile.jpg`, the service worker could decide to respond to that request with cached content rather than having the request go through to the actual server. There is a lot more than this to service workers, but in order to get a basic PWA functioning, we will only need to perform a few basic tasks.

Enabling PWA Support

In order to enable PWA support for our applications through a service worker, we are going to be using **Angular Service Workers**. Service workers are complicated and tricky to work with, but Angular has integrated them into the tooling and framework in such a way that we can pretty much just run a single command and be done with it. The following steps will walk you through doing this.

1. Add the Service Worker

Adding the service worker is as simple as running the following command in your existing Ionic project:

```
ng add @angular/pwa --project app
```

The `app` at the end of this command is the name of your project as defined in the **angular.json** file. By default, this will be `app` but if your project has a different name make sure to change that here (you probably won't need to do this). To reiterate, **don't replace**

"app" with your projects name unless it is also not called "app" in angular.json.

Once you run this command, you will see various files being created and updated:

```
Installed packages for tooling via npm.
CREATE ngsw-config.json (624 bytes)
CREATE src/manifest.webmanifest (1330 bytes)
CREATE src/assets/icons/icon-128x128.png (1253 bytes)
CREATE src/assets/icons/icon-144x144.png (1394 bytes)
CREATE src/assets/icons/icon-152x152.png (1427 bytes)
CREATE src/assets/icons/icon-192x192.png (1790 bytes)
CREATE src/assets/icons/icon-384x384.png (3557 bytes)
CREATE src/assets/icons/icon-512x512.png (5008 bytes)
CREATE src/assets/icons/icon-72x72.png (792 bytes)
CREATE src/assets/icons/icon-96x96.png (958 bytes)
UPDATE angular.json (5433 bytes)
UPDATE package.json (1769 bytes)
UPDATE src/app/app.module.ts (995 bytes)
UPDATE src/index.html (909 bytes)
```

The command automatically does the following for you:

- Creates the configuration file for the service worker
- Creates the various icons that are required for the home screen
- Updates the **angular.json** file to enable service worker support
- Updates the **package.json** file to include the `@angular/pwa` package

- Imports the service worker functionality into the root module file
- Adds the required meta tags to the **index.html** file

If you take a look at the **ngsw-config.json** file, you will see something like this:

```
{
  "$schema": "./node_modules/@angular/service-
worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
```

```
      "resources": {
        "files": [
          "/assets/**",
          "/*.
(eot|svg|cur|jpg|png|webp|gif|otf|ttf|woff|woff2|ani)"
        ]
      }
    }
  ]
}
```

This defines the behaviour of the service worker and includes all of the files that we want to be cached (which is what will enable the application to be accessed even while offline). All of the code we would need for a standard Ionic application is already being included here.

If you take a look at the **app.module.ts** file, you will see the following new import:

```
  imports: [
    ...
    ServiceWorkerModule.register('ngsw-worker.js', { enabled:
environment.production })
  ],
```

This is what handles registering the service worker, and enables it for production builds.

2. Creating a Production Build

To create a production build of your Ionic PWA, you can just run the following command:

```
ionic build --prod
```

This will run the build process, and output the result to your **www** folder. The contents of this folder are your completed PWA (but remember, if you serve your application again or create another build the contents of this folder will be overwritten). You could now just host the contents of this folder somewhere, and your PWA would be live (we will be walking through how to do this with Firebase Hosting in a moment) - it's basically just a standard website (but cooler) at this point, and you can host the files in any way you would a normal website. The only difference is that the files do need to be hosted on a web server, you can't serve these files over the filesystem (i.e. just by opening up the **index.html** file in your browser).

A good way to test these builds is to use a utility like serve. This is kind of the same concept as `ionic serve` except that it lets you create a local web server to serve the contents of any directory on your computer. If you have this installed, you can then just navigate to the **www** folder of your project:

```
cd www
```

and then serve the contents of that folder:

```
serve -p 8080
```

You could then go to:

```
http://localhost:8080
```

In your browser to view your PWA. You can also easily test offline behaviour by going to the `Network` tab in the debugging tools and checking the `Offline` box to simulate an offline state. If you refresh the application, you would see something like this:

You can see that all of the requests are being fulfilled by the service worker (the size column says "from service worker"). Your content will be served by the service worker even when you are online which can greatly improve loading speeds, but this is also what

allows the application to continue to function even when offline.

3. Updating the Application

One of the tricky aspects of serving cached content from service workers is handling updates to the application. If the content has not changed then we want it served from the service worker, but if the content has changed then we would want to pull in those updates from the network.

Fortunately, Angular handles this for us. Whenever the application is loaded it will check for changes to the app by inspecting the **ngsw.json** file. If there are changes, then the new content will be downloaded and cached. The downside to this is that this happens *after* the application has loaded, and the user won't see the new content until they refresh the application. This may not even be a problem, but if it is, you can listen for updates using the SwUpdate service and prompt the user to refresh the application when an update is found (or you could even force a reload if you really need to).

Hosting the PWA with Firebase

At the end of the previous section, we have all of the files necessary for our PWA inside of the **www** folder. You can host those files anywhere you like, as long as people can access it via a URL, e.g:

```
https://mywebsite.com
```

or you could have it on a subdomain of your website:

```
https://mobile.mywebsite.com
```

As long as the user can access those files via a URL through a web browser, it doesn't matter how or where you host it, they will be able to view your PWA. However, in this section, we are going to walk through hosting specifically with Firebase Hosting.

NOTE: Depending on how the server is configured, a PWA may not work out of the box with every host. The host must support single page applications, i.e. they must have a method for routing all requests through the `index.html` file. If you have a route like `https://example.com/checklists/25` in your application, you don't want the server to attempt to serve a file that lives inside of the `checklists` folder and is called `25`. This request needs to be sent to the `index.html` file, and then the application decides what to do with the route.

Firebase Hosting is very friendly for PWA deployment. There is no need to set up a complicated deploy process, you can just run a single command to upload the latest version of your application, and they have a very generous free tier.

At the time of writing this, the free hosting plan offers 10GB of storage and 360MB/day transfer. You even get a free SSL certificate so that you can serve your application over the https protocol without having to configure the certificate yourself. You can also apply your own custom domain name on the free tier.

It's important to note that you do not need to use other Firebase services in order to use Firebase Hosting. Firebase is a popular choice for backend development, and it is often used in Ionic applications. Hosting is a different service, and it doesn't require that you are building a "Firebase app".

To get started, you might first want to take a look at the available plans, choose one that suits you, and create an account if you have not already.

If you have already signed up then you can make your way directly to console.firebase.google.com. From here you will need to create a project, or use an existing project, to add Firebase hosting to.

Once you have done this, you will be taken to the Firebase dashboard where you can access everything associated with the project (authentication, cloud functions, database, etc.). We only need hosting in this instance, so you should click the **Hosting** option from the menu.

Click **Get Started** and it will prompt you through setting up the project. If you have not already done so, you will need to install the `firebase-tools` package globally:

```
npm install -g firebase-tools
```

The `-g` flag means that it will be installed globally on your computer, not just in the current project. Once you have installed it once, you won't need to reinstall it for future projects (except to update the `firebase-tools` package occasionally).

You will then need to log in to the Firebase tools:

```
firebase login
```

and then from the root of your Ionic project, you will need to run the following command:

```
firebase init
```

You will then be presented with some options. Use the down arrow to navigate to the **Hosting** option, and then make sure to hit **SPACE** and then **ENTER**. You will then be given a list of projects that you can associate this Firebase project with – select the appropriate one and then hit **ENTER**.

Now we will need to go through configuring the project. Since the **www** folder contains our built code for the Progressive Web Application, this is the folder that we want to set as the **public directory** (i.e. the folder that we want to be uploaded to Firebase hosting). If asked, you should configure your application as a single-page app, but **do not** overwrite the existing `index.html` file.

You will find two new files in your project: **firebase.json** and **.firebaserc**. If you need to, you can make changes to your projects configuration in the **firebase.json** file.

If you haven't already, build a production version of your application with:

```
ionic build --prod
```

IMPORTANT: Make sure to always run this command before deploying to Firebase hosting.

and then:

```
firebase deploy
```

Once you run the `deploy` command, you will be given the **Hosting URL** that you can use to access your PWA hosted on Firebase! You will be able to use it just like you can through `ionic serve`, except you will be able to access it from any mobile device and add it to the home screen. If you've specified the standalone display option in your `manifest.webmanifest` file for the PWA, the application will launch from your home screen just like a native application would (and you will also be able to access it offline).

NOTE: You will also want to update the `manifest.webmanifest` file with information specific to your application (e.g. its name, theme colours, icons, and so on).

Conclusion

If you've made it through this entire book (and even if you haven't yet) I would like to give you a huge...

Thank you!

Not only have you invested your time and money into learning how to build mobile apps with Ionic (which I think is a very wise choice), you've also invested time and money into me personally. I love teaching developers mobile development and it means a lot to me to know that you trust in my abilities enough to buy this book, and by doing that you're also allowing me to invest more time into creating even more content for mobile developers.

I assume if you've completed this book then you're probably familiar with my blog, but if you're not, make sure to check it out. I frequently release free content there, and if you're looking to expand your Ionic skills even more, there's plenty to check out. You can also subscribe to my YouTube Channel.

When you are ready to jump into more advanced Ionic/Angular content, you might be interested in checking out my advanced course for Ionic developers: Elite Ionic. If you've bought one of the smaller packages for this book and would like to upgrade, feel free to send me an email and I can sort it out for you.

If you are interested in learning more about how to create advanced gestures and animations in your application (think things like Tinder swipe cards, fancy screen transitions, custom interactions) you might also like to check out Advanced Animations &

Interactions with Ionic.

I hope you enjoyed this book and have taken a lot away from it. I'm always looking for feedback on how to continually improve though so please send me an email with any feedback you have.

If you need help with anything related to mobile development feel free to get in touch. I respond to as many requests for help as I can but I receive a lot of emails so I can't always get to them all. If you want more personal and guaranteed support, I also offer consulting services.

Also, feel free to send me an email just to let me know what you're working on, I always like hearing what my fellow developers are up to!