



JOSHUA MORONY

BUILDING MOBILE APPS WITH IONIC & ANGULAR



Changelog

Version 21 (this version) - 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 - updated for Ionic 2.2.0

- Updated Storage implementation
- Configuration update for WKWebView plugin
- Updated Fliqs theme to work with grid changes (added nowrap attribute)
- Minor bug fixes and improvements

Version 14 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: Local and Remote Backend with PouchDB and Cloudant](#)
57. [Lesson 7: Styling & Animations](#)
58. [Camper Chat Conclusion](#)

Testing and Debugging

59. [Testing & Debugging](#)

Building and Submitting

60. [Preparing Assets](#)

61. [Building for iOS and Distributing to the Apple App Store](#)

62. [Building for Android and Distributing to Google Play](#)

63. [Creating iOS Certificates on Windows](#)

64. [Building for iOS and Android Using Ionic Package \(Coming Soon\)](#)

65. [Publishing as a PWA \(Progressive Web Application\) with Firebase](#)

Conclusion

66. [Conclusion](#)

Introduction

Welcome!

Hello and welcome to **Building Mobile Apps with Ionic & 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 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 experience with Ionic.

This book does not contain an introduction to HTML, CSS, and JavaScript, though. 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 HTML & CSS](#)
- [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 copying 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.

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 (and all of the code in this book shouldn't require any changes until Ionic 5 rolls around). 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 small 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 unless there is a significant difference.

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 is now 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 elements that work anywhere on the web.

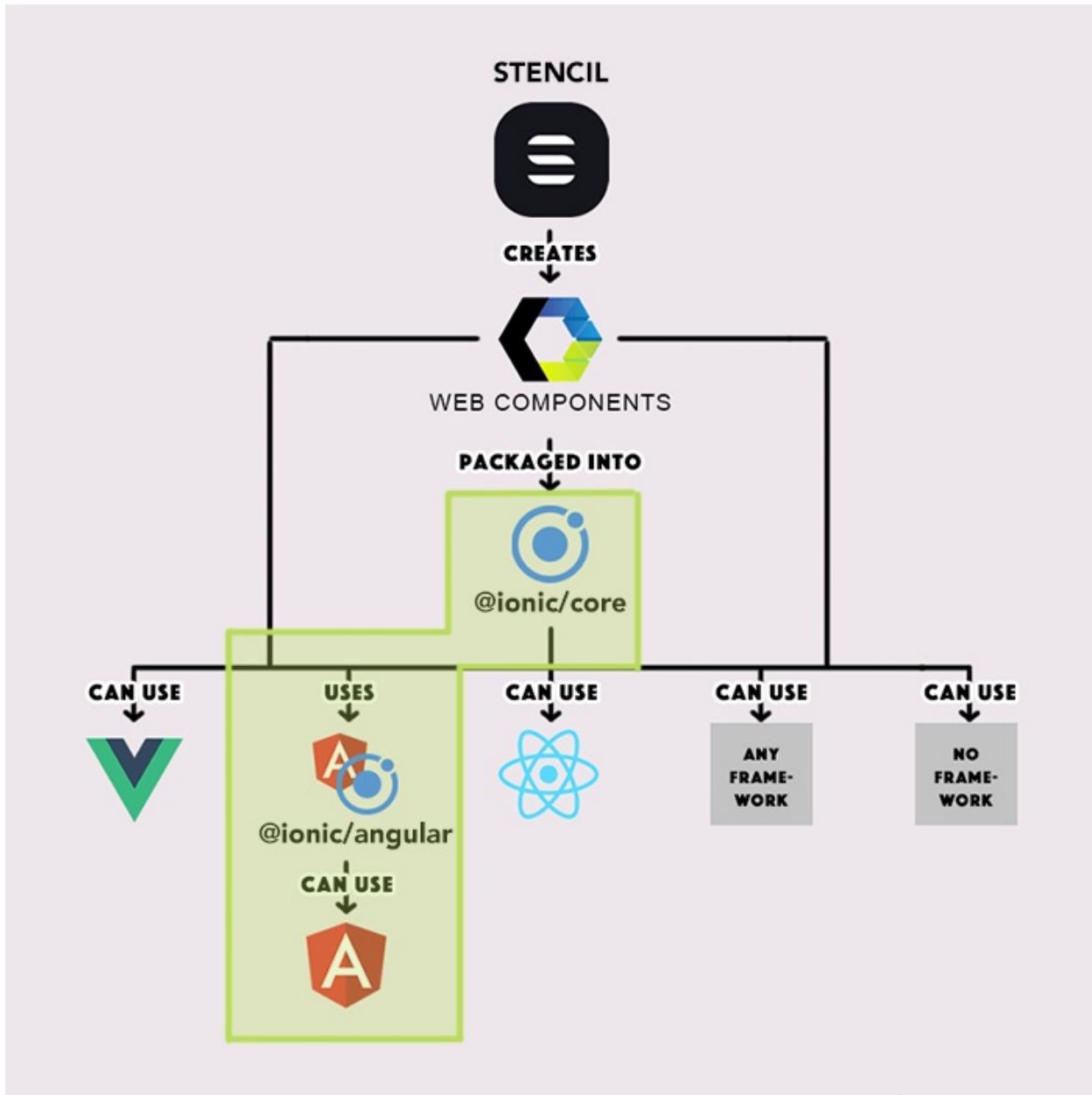
We can still use Ionic/Angular just as we could before, but because of web components now we can also use those same Ionic components in other frameworks (like Vue or React), or with no framework at all.

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
- Stencil
- 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 it is going on.

Stencil is a tool that creates web components, and the Ionic team has created a bunch of

web components with Stencil which has become **Ionic Core**. Ionic Core contains the set of web components that make up Ionic's user interface library (lists, buttons, and so on). You will use this core set of components no matter where you are using Ionic, but you don't need to use or understand Stencil itself.

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, **you don't need to know how to use Stencil in order to use Ionic.**

The bundle of web components that is "Ionic Core" can then be used by any framework, and it can also be used without a framework at all. There is a special library called `@ionic/angular` which is the Ionic library framework 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. This means that it is a little "nicer" to use Ionic with Angular, but there is no reason you can't use Ionic's core components in whatever framework you like.

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 organised structure to work within, as well as a bunch of built in features to make development easier.

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
- 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 combined with Angular is the best option for most people in the general case, including those who have used Ionic/Angular in the past and for those who are new to mobile development.

Using Ionic with a Different Framework

Although I think that there are a lot of benefits to using Angular over other frameworks for Ionic, 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.

In general, I would recommend this approach if you already have a preference for a particular framework, like Vue or React, and you are more comfortable with using that. You may miss out on the benefit of having a framework specific library like `@ionic/angular`, but you gain the benefit of being able to work in an environment that you are more comfortable with.

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 is heavily optimised and doesn't require the

overhead of a framework.

I would recommend this approach to people who are willing to put in more development effort to really push the limits of performance, and especially if you are targetting users who may have low-end devices or you need to cater for slow network speeds or expensive data.

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 tappable 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-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">  
          
          
          
          
      </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;
});

displayUserName();
```

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 `displayUserName` 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 `displayUserName` 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.

NOTE: There is an alternate syntax available for handling promises with `async` and `await`. The same underlying concept is still the same, and promises are still used, but you may find that using the `async/await` syntax can lead to "simpler" code. If you are familiar with this syntax, you may use it if you wish instead of the "standard" promise syntax that is used in this book. There isn't really a right and wrong approach here, it just comes down to preference and I find that the standard promise syntax is easier to understand and explain for beginners.

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.

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 targetting (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 Progressive Web Apps**. Capacitor wants to be a runtime layer that allows you to build an application 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 described 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 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)

```
npm install --save @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:

```
Camera.getPhoto().then((photo) => {
  console.log(photo);
}, (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!

All Capacitor APIs work in more or less this format:

```
SomePlugin.someMethod().then((result) => {
  console.log(result);
}, (err) => {
  console.log(err);
})
```

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) - requires macOS
- Install Android Studio (if you want to build Android applications)

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 have a service called **Ionic Package** that can perform the iOS builds for you (so that you don't need a Mac yourself), and support for Capacitor in this service will likely be available soon.

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` library (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 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 10.15.0
```

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
```

or

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

NOTE: It is better to install Ionic using the first command. If you run into permission issues you should try to solve those first before "brute forcing" the install by elevating privileges with 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
```

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.

I coded using Sublime Text for a long time, but when I switched to VS Code it was definitely one of those "Why didn't I do this earlier?" moments. The TypeScript support is just such a perfect fit for Ionic/Angular applications.

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.

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
```

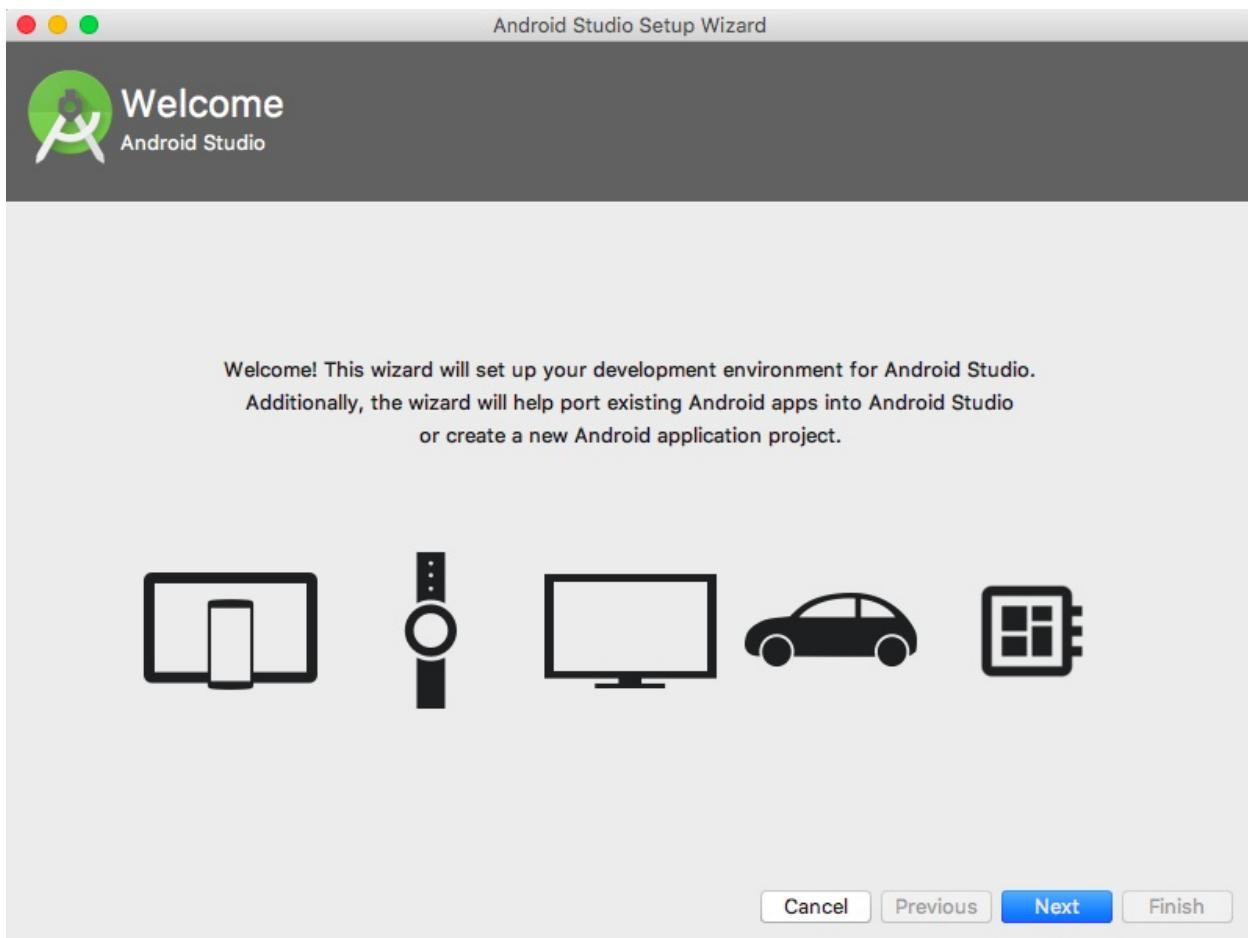
Once you have installed Xcode, make sure to open it at least once so that it can install the necessary components.

Installing Android Studio (Optional)

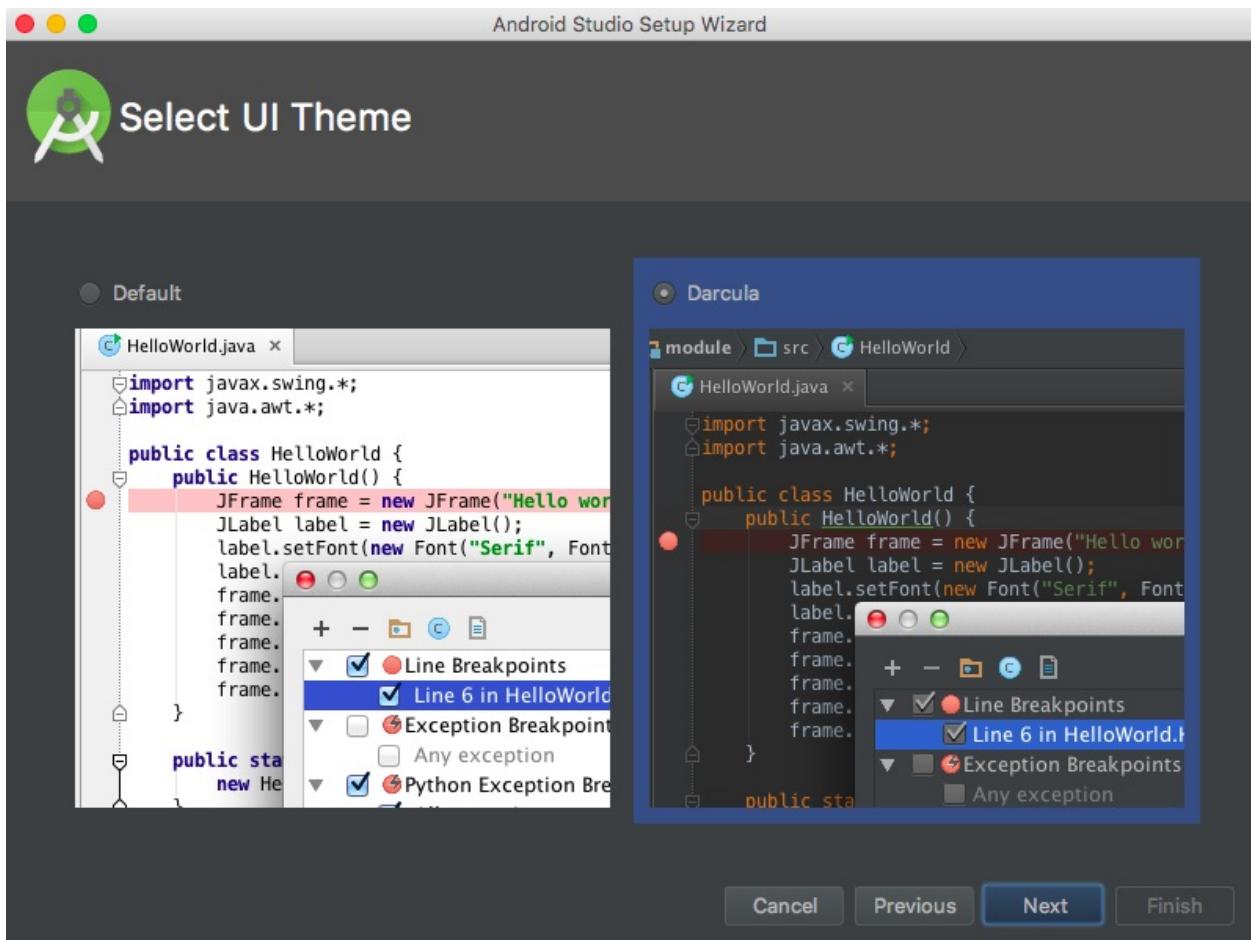
We will need to use [Android Studio](#) to build native Android applications and submit them to Google Play. If you do not already have Android Studio, you should install it by visiting [this website](#).

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

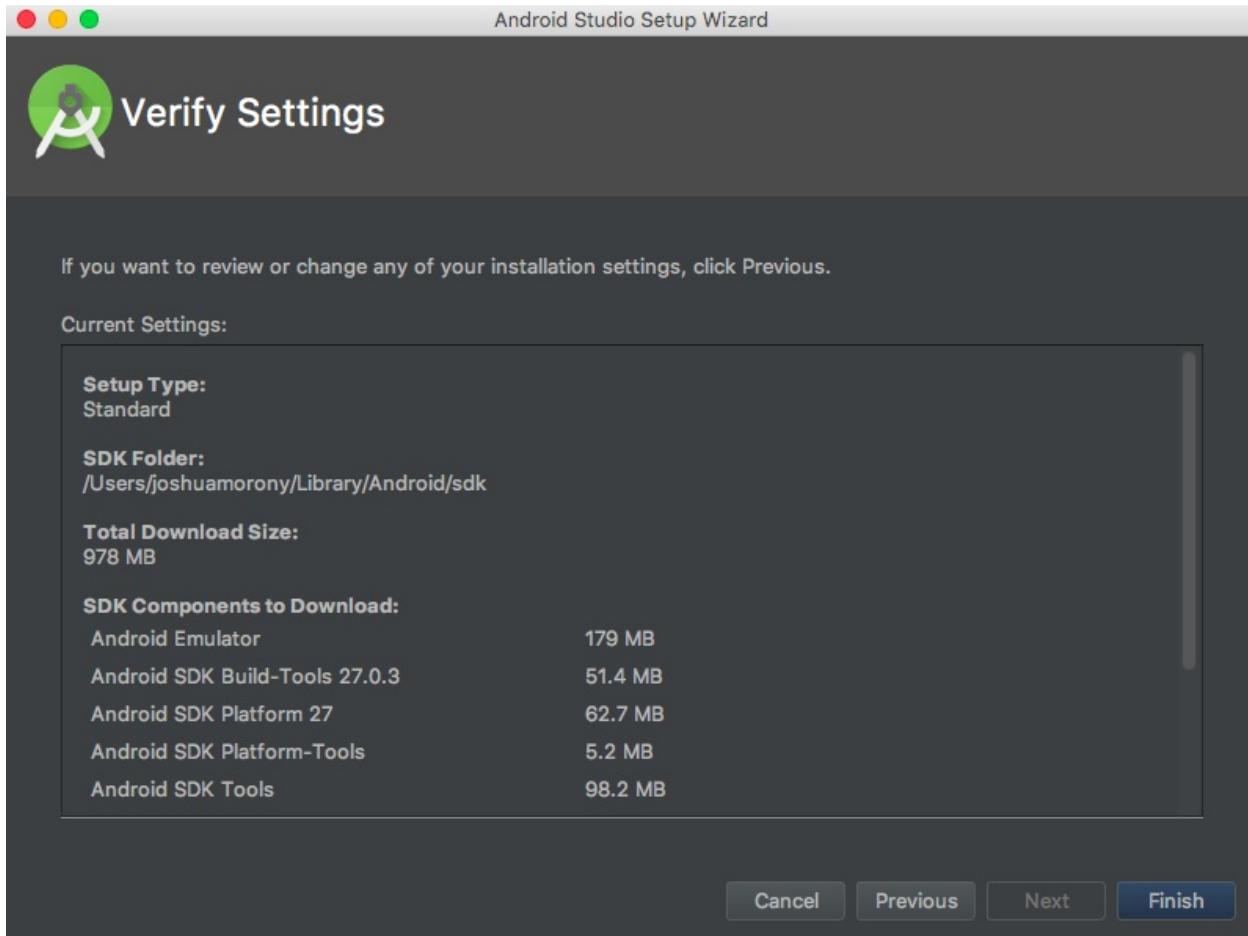
Once it is installed, you should open it. The first time you open Android Studio you will be taken through a setup wizard:



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.

NOTE: Whilst Ionic 4 is in beta, you will need to manually specify the angular project type when creating a project.

> **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 come in very useful if they closely match the structure of the application that you want to build.

Your project will begin generating, and you may also be asked some additional questions:

- If asked if you want to install the free **Ionic Appflow SDK** say **No** - we will not be making use of Ionic Appflow in this book (Appflow provides premium services for Ionic applications)

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 HTML5 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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

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
```

or

```
sudo npm install -g ionic
```

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:

```
▶ e2e
▶ node_modules
◀ src
  ▲ app
    ▶ components
    ▲ pages
      ▲ home
        TS home.module.ts
        ◁ home.page.html
        ⚡ home.page.scss
        TS home.page.spec.ts
        TS home.page.ts
        TS app-routing.module.ts
        ◁ app.component.html
        TS app.component.spec.ts
        TS app.component.ts
        TS app.module.ts
    ▶ assets
    ▶ environments
      ▲ theme
        ⚡ variables.scss
        ⚡ global.scss
        ◁ index.html
        K karma.conf.js
        TS main.ts
        TS polyfills.ts
        TS test.ts
        {} tsconfig.app.json
        {} tsconfig.spec.json
    ▶ www
      ♦ .gitignore
      {} angular.json
      ⓘ ionic.config.json
      {} package-lock.json
      {} package.json
      {} tsconfig.json
      {} tslint.json
```

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 we 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

Ionic used to have a specific "pages" folder that contained all of the "pages" for the application, but now these "pages" live in 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 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 --save
```

It is important to include the `--save` flag, because this will add the dependency to your **package.json** file (as opposed to just installing it). 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.

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 --save @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 pages/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 (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 --save
```

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 --save
```

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@beta --save
```

NOTE: Whilst Ionic Native is in beta, it is important that you install Ionic Native plugins

using the @beta version. Just add @beta to the end of the package. 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.

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 come in 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's 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  
}  
  
isOld(){  
    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:

```
let 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 `isOld`. 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:

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

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

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 (he may be old, but I'm sure he is wise).

Classes in Ionic

So, 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.isOld = function() {  
    return this.age > 70;  
};  
  
var david = new Person('david', 72);  
console.log(david.isOld());
```

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-item tappable>
```

or:

```
<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:

```
this.dataService.getData().then((data) => {  
  console.log(data);  
});
```

Notice that we use `then()` 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:

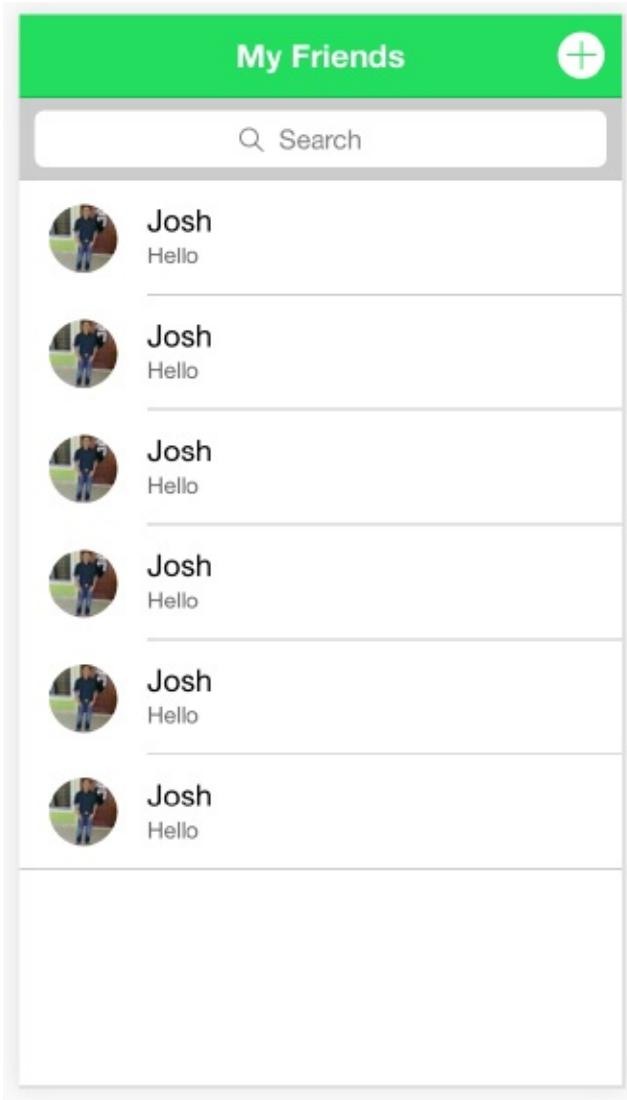
```
<ion-header>
  <ion-toolbar color="secondary">
    <ion-title>
      My Friends
    </ion-title>

    <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>
```

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

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



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">
```

or

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

`*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">
```

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]="show MyClass" slot="start">
```

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 `show MyClass` evaluates to be true. This can come in really handy, 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.

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 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.

So, 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](#).

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:

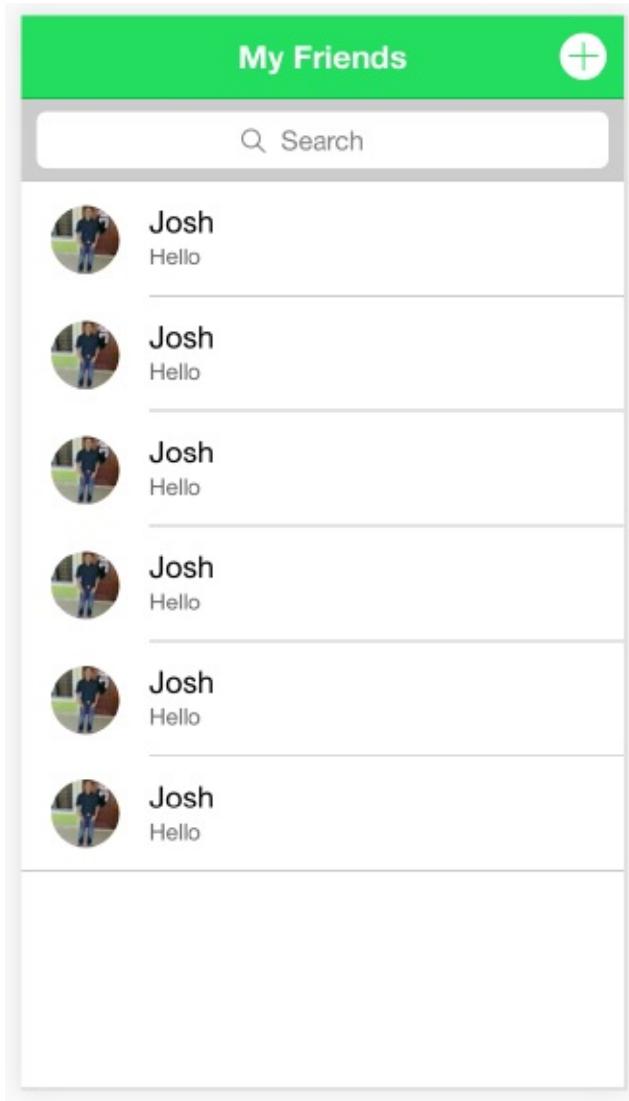
```
<ion-icon name="heart"></ion-icon>
```

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:



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:



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.

I certainly don't claim to be some design guru, but I think the themes I've created for these applications are visually pleasing and help give the apps some character. 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, which is primarily due to the fact that it uses **SASS** and **CSS4 Variables**. SASS and CSS4 variables are not 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 CSS4 variables instead of SASS variables (which was the primary feature SASS was used for). One of the primary benefits of CSS4 variables is that they are native to the browser, and don't require precompiling like SASS variables do. Although SASS is still used in Ionic, and you can use all of the features of SASS as you wish, you probably won't really need to know much about how it works. Most people will just tinker with a few CSS4 variables, rather than making use of other SASS features.

A CSS4 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 create these variables yourself, but rather overwrite the default Ionic CSS4 variables that look like this:

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

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 CSS4 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 CSS4 variable.
- **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 one 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, CSS4 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 a the toolbar I could do this:

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

Keep in mind that attributes are not limited to just changing the colour of elements, some attributes will affect different styles. For example, if we wanted to remove the borders from

a list component we could do this:

```
<ion-list no-lines></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 attributes, so make sure to poke around the documentation when you are using Ionic's built-in components. The `no-lines` attribute is a real 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. CSS4 Variables

The next method you can use to control the style of your application is to change the default CSS4 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 CSS4 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, 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 CSS4 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:

```
app-home {  
  --ion-background-color: #000;  
}
```

The great thing about editing these default CSS4 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.

3. Custom Styles

We've talked about using CSS4 variables and applying attributes 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.

Of course, 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 attributes and CSS4 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.

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:

```
.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 CSS4 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 CSS (e.g. `ion-item`), but you can not target anything inside of the web component. Ionic will eventually have a list of all of the CSS4 variables that the various components are utilising in the documentation. For now, if you want some advice on how to go about styling the internals of the web components, check out [my blog post](#).

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 will be covering is something that was introduced in the Ionic 4 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 web components](#). 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. Angular is a bit of a special case in this regard - since Ionic and Angular have been tightly integrated in the past, the Ionic specific push/pop navigation already exists and so it is being used by people in their applications currently (which, I believe, is why the Ionic team have decided to keep that option around).

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 completley 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). Notice that we use square brackets around **routerLink** here because we don't want to use the literal string:

```
"/detail/' + item.id "
```

as the **routerLink**, we want that expression to be evaluated first. By using the square brackets around the property, it will know that we want to add the string `/detail/` to whatever `item.id` is. If `item.id` evaluates to be 12 then the final link would become:

```
/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 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. There is no harm in using lazy loading, so I would encourage everybody to use it 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:
    './pages/login/login.module#LoginModule' },
  { path: 'home', loadChildren:
    './pages/home/home.module#HomeModule' },
  { path: 'detail/:id', loadChildren:
    './pages/detail/detail.module#DetailModule' },
  { 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), and we also need to supply the name of the module class by appending it to the end with a # (i.e. the name of the exported class in the module file).

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

```
'./pages/home/home.module#HomeModule'
```

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

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

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

@NgModule({
  imports: [
    CommonModule,
    IonicModule,
    HomePageRoutingModule
  ],
  declarations: [HomePage],
```

```
entryComponents: [],
bootstrap: [HomePage],
})

export class HomeModule {}
```

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 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>

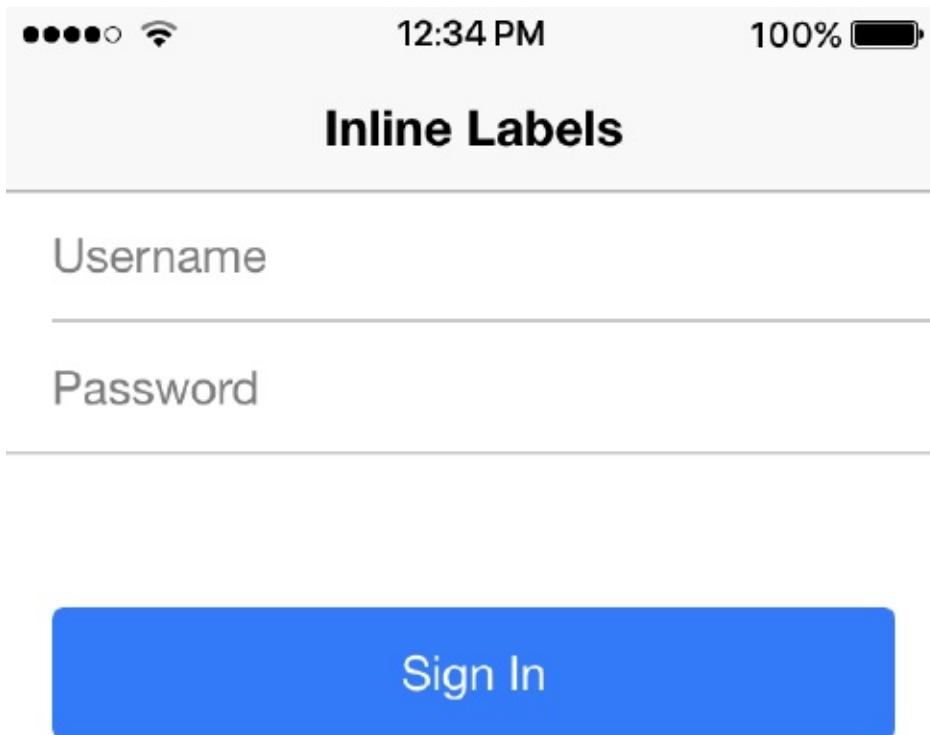
  <ion-item>
    <ion-label>Username</ion-label>
    <ion-input type="text"></ion-input>
  </ion-item>

  <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:

```
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` function when it is clicked, we could add that to our class:

```
logValue(){
  console.log(this.myValue);
}
```

This function 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 function, 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` function.

```
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.

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)

HTML5 applications run in the 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 reauthenticate.

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');

let 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 --save
```

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 its 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 add the 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(): 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 add a `.then()`. For example, if I wanted to use the `getData()` function defined above I would need to do:

```
this.getData().then((data) => {  
  console.log(data); // This will contain the checklists  
});  
  
console.log(data); // This will NOT work, because it is outside  
of the promise handler
```

Even though the second `console.log` is triggered after the promise, it will still execute before the promise resolves (the promise has finished fetching the data from storage), so it won't work.

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:

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

let 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:

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];
```

```
let 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:

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

let newArray = oldArray.filter((entry) => {
  return entry.indexOf('.png') > -1;
}).map((entry) => {
  return 'http://www.example.com/' + entry;
});
```

So 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 cover 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:

```
getFromSlowServer().then((data) => {
  console.log(data);
});
```

We call the `then` method which a Promise provides, which basically says "Once you have the data from the server, do this with it". In this case, we are passing the data returned to a function where we log it out to the console. 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` method 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()
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:

```
let 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 **Http** 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. So 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 **Http** 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){
```

```
}
```

```
}
```

Since we have injected the HttpClient service into our constructor and made it available through `this.http` by using the `private` keyword, we can now make use of it anywhere in this 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. So 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(1500),  
  distinctUntilChanged()  
) .subscribe((subreddit) => {  
  
  if(subreddit.length > 0){  
    this.redditService.changeSubreddit(subreddit);  
  }  
  
})
```

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 1.5 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. 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 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");
});
```

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 --save @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 --save
```

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';
```

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';
```

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);
});
```

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;
```

```
Geolocation.getCurrentPosition().then(position) => {
    console.log(position);
};
```

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. 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. For an example of how to do this, I would recommend reading: [Using Cordova Plugins that Require Install Variables with Capacitor](#)

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 too advanced for this book, however, I have written a blog post on this topic if you would like an introduction to [creating your own Capacitor plugins](#). 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 course - no matter which of the packages you purchased, you will have access to this lesson. 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. 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. 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 to the car, that the gas is off, and that the brakes and indicators are working.

So, 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.

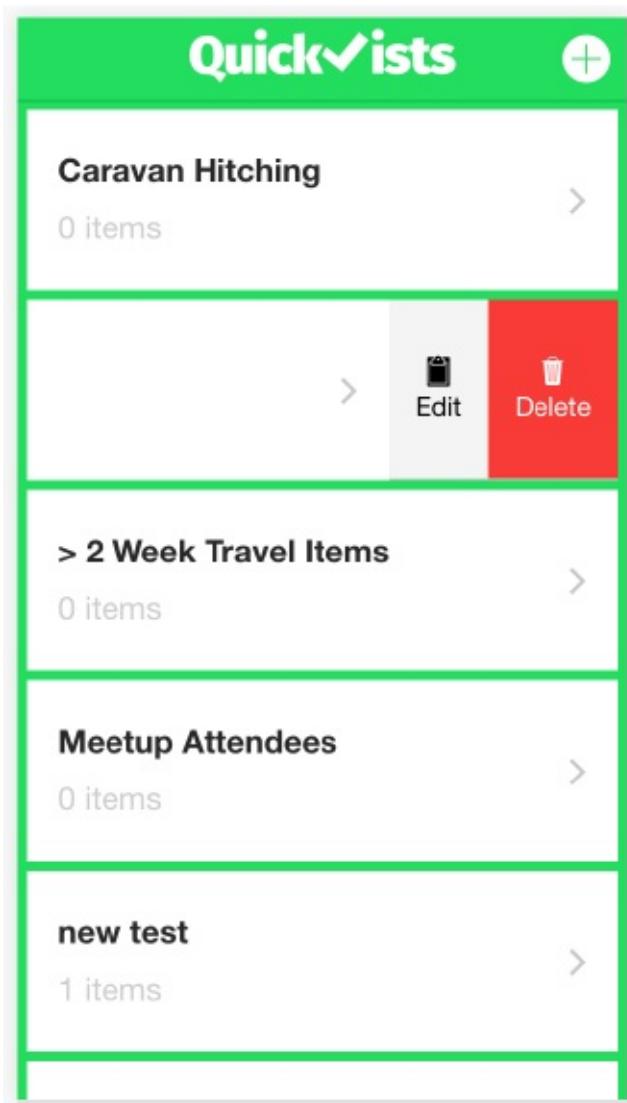
As I mentioned, this application covers similar concepts to what a traditional todo application would. Specifically, though, the main concepts we will cover are:

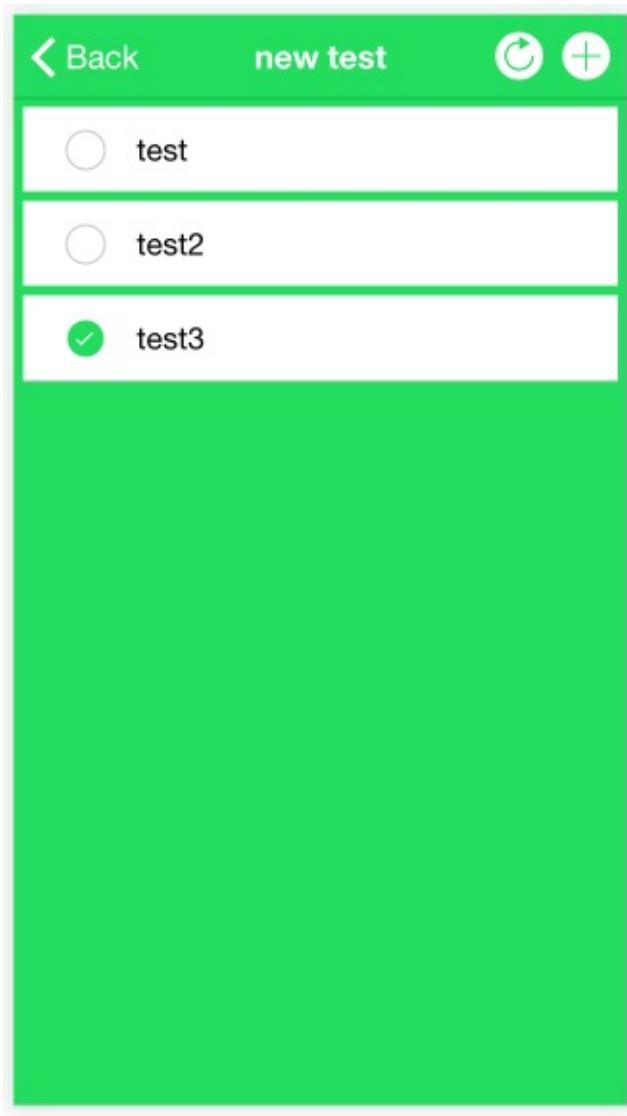
- Complex Lists
- Data Types / Interfaces
- Forms and User Input
- Simple Navigation
- Passing Data Between Pages
- Creating, Reading, Updating and Deleting Data
- Data Storage and Retrieval
- Theming

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
- All data will be remembered upon returning to the application (including the completion state of checklist items)

and here's a couple of screenshots to put everything in context.





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 & Theming](#)

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
```

or

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

before you continue. If you run into any trouble installing Ionic or generating new projects, make sure that you have the most recent [NodeJS LTS](#) version installed. After you have that installed, you should also run the following command:

```
npm uninstall -g ionic
```

before attempting to install again.

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 repurpose 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 asked, **do not** integrate the Ionic Appflow SDK.

> **Make the new project your current working directory by running the following command:**

```
cd quicklists
```

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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

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 data service in this application to help us out. This will handle saving the checklist data into storage and retrieving them from storage.

> Run the following command to generate a ChecklistData service:

```
ionic g service services/ChecklistData
```

NOTE: Notice that we use services/ChecklistData instead of just ChecklistData this time. This is because we want to generate all of our services inside of a folder called services.

Create the Interface

We will eventually be defining a custom "Interface" to represent our Checklist and ChecklistItem data types. 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 checklists.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 { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: '/checklists', pathMatch: 'full' },
  { path: 'intro', loadChildren:
    './intro/intro.module#IntroPageModule' },
  { path: 'checklists', loadChildren:
    './home/home.module#HomePageModule' },
  { path: 'checklists/:id', loadChildren:
    './checklist/checklist.module#ChecklistPageModule' }
```

```
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  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

Capacitor is not included by default in our applications, so we need to set it up. You should still complete this step even if you don't intend to build for iOS and Android because we will also be using Capacitor for the web platform.

To set up Capacitor in an Ionic application, all you need to do is run a single command.

> Run the following command to enable Capacitor:

```
ionic integrations enable capacitor
```

This will handle setting up the Capacitor CLI and Capacitor Library in your project.

Keep in mind that this will give your application a default 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 now. So, you may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`.

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. 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 application code and dependencies over at any time by running:

```
ionic cap sync
```

When you make changes to your application build, you will need to run this command to copy the changes over to Capacitor.

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'
})
export class AppComponent {

  constructor() {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });
  }
}
```

All we are doing here is hiding the StatusBar but you can use this API to style the Status Bar however you like. 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 likely to get wiped by the operating system).

To enable Ionic storage, you need to install it.

> **Install Ionic Storage with the following command:**

```
npm install @ionic/storage --save
```

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 { RouterModule, RouteReuseStrategy, Routes } 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 --save
```

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 even get issues that don't seem to make sense, try stopping your application from serving with `Ctrl + C` and then run `ionic serve` again.

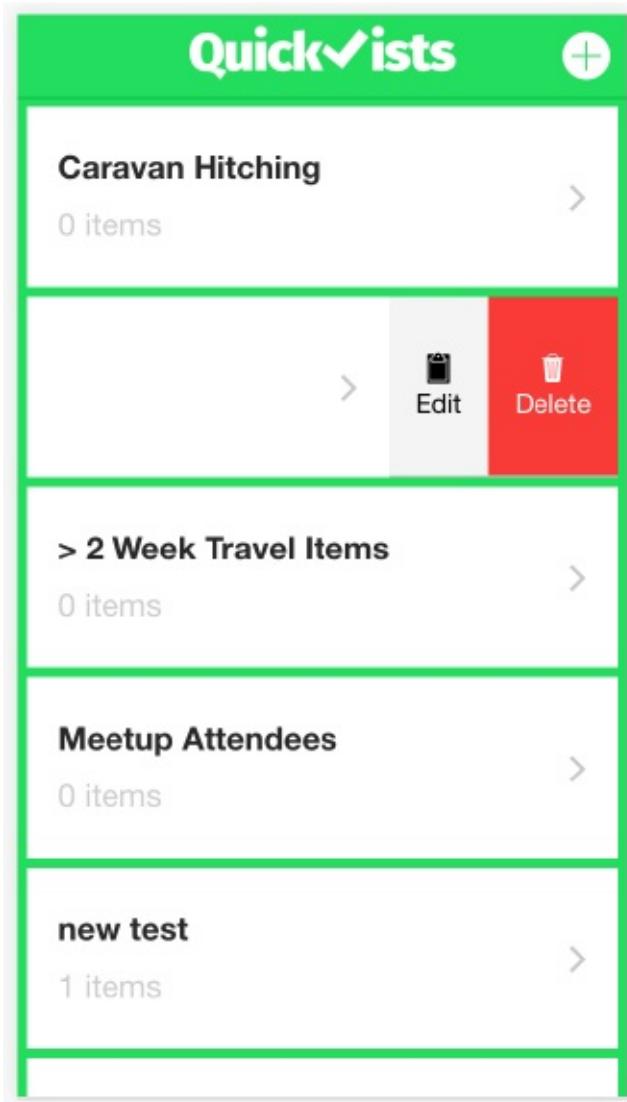
Lesson 3: Basic Layout

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 course as modular as possible so that you can build the applications that interest you first, and aren't forced to follow a particular order. Since this is the first application though, 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:



It's 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 sliding list item component Ionic provides, which allows us to specify some content that will be displayed when the user swipes the 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="secondary">
    <ion-title>
      
    </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="addChecklist()"><ion-icon slot="icon-only" name="add-circle"></ion-icon></ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content>

  <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-sliding>
  </ion-list>
</ion-content>
```

```

<ion-item-options side="end">
  <ion-item-option color="light" expandable
(click)="renameChecklist(checklist)"><ion-icon slot="icon-only"
name="clipboard"></ion-icon></ion-item-option>
  <ion-item-option color="danger" expandable
(click)="removeChecklist(checklist)"><ion-icon slot="icon-only"
name="trash"></ion-icon></ion-item-option>
</ion-item-options>

</ion-item-sliding>

</ion-list>
</ion-content>

```

Let's start off by discussing the `<ion-header>` section:

```

<ion-header>
  <ion-toolbar color="secondary">
    <ion-title>
      
    </ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="addChecklist()"><ion-icon slot="icon-only"
name="add-circle"></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` attribute to the toolbar to style it with our secondary colour, this is defined by default but we can also modify it to be whatever we like (we covered this in the **Styling & Themeing** lesson earlier in the book). 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>` to create a button in the toolbar. By using the `end` slot, the buttons will be placed on the right-hand side of the toolbar.

Finally, we have the button itself inside of `<ion-buttons>`. This button uses a circle 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 created 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).

Let's move onto the content and list section now:

```
<ion-content>
```

```

<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" expandable
        (click)="renameChecklist(checklist)"><ion-icon slot="icon-only"
        name="clipboard"></ion-icon></ion-item-option>
      <ion-item-option color="danger" expandable
        (click)="removeChecklist(checklist)"><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 header section with 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 the `lines="none"` to the `<ion-list>`. Just like the `secondary` attribute, we added to the toolbar, this attribute 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.

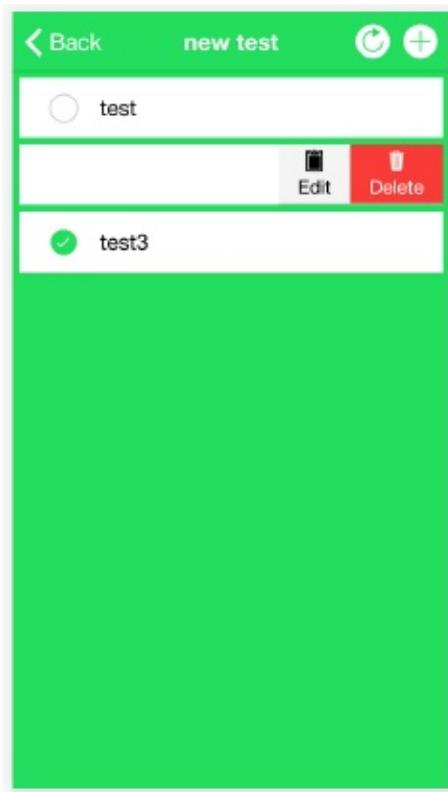
The first block of code inside of `<ion-item-sliding>` is the normal `<ion-item>` component. We have also attached a `routerLink` to our `<ion-item>` which, if you remember from the section on navigation, allows us to link to a route in our application. In this case, we are going to the checklists route, 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). 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:



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>
  <ion-toolbar color="secondary">
    <ion-title>
      CHECKLIST TITLE
    </ion-title>
  </ion-toolbar>
</ion-header>
```

```
</ion-title>
<ion-buttons slot="start">
  <ion-back-button defaultHref="/checklists"></ion-back-
button>
</ion-buttons>
<ion-buttons slot="end">
  <ion-button (click)="addItem()"><ion-icon slot="icon-
only" name="add-circle"></ion-icon></ion-button>
</ion-buttons>
</ion-toolbar>
</ion-header>

<ion-content>

<ion-list lines="none">

<ion-item-sliding>

  <ion-item>
    <ion-label>ITEM TITLE</ion-label>
    <ion-checkbox [checked]="item.checked"
      (ionChange)="toggleItem(item)"></ion-checkbox>
  </ion-item>

  <ion-item-options side="end">
    <ion-item-option expandable color="light"
      (click)="renameItem(item)"><ion-icon slot="icon-only"
      name="clipboard"></ion-icon></ion-item-option>
```

```
<ion-item-option expandable 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. We also supply a defaultHref, this way if the page is refreshed (and the history is lost) the back button will still know what page it needs to navigate the user back to.

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>` as the main item instead of the title of the checklist and when it 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 **item.checked**. 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:

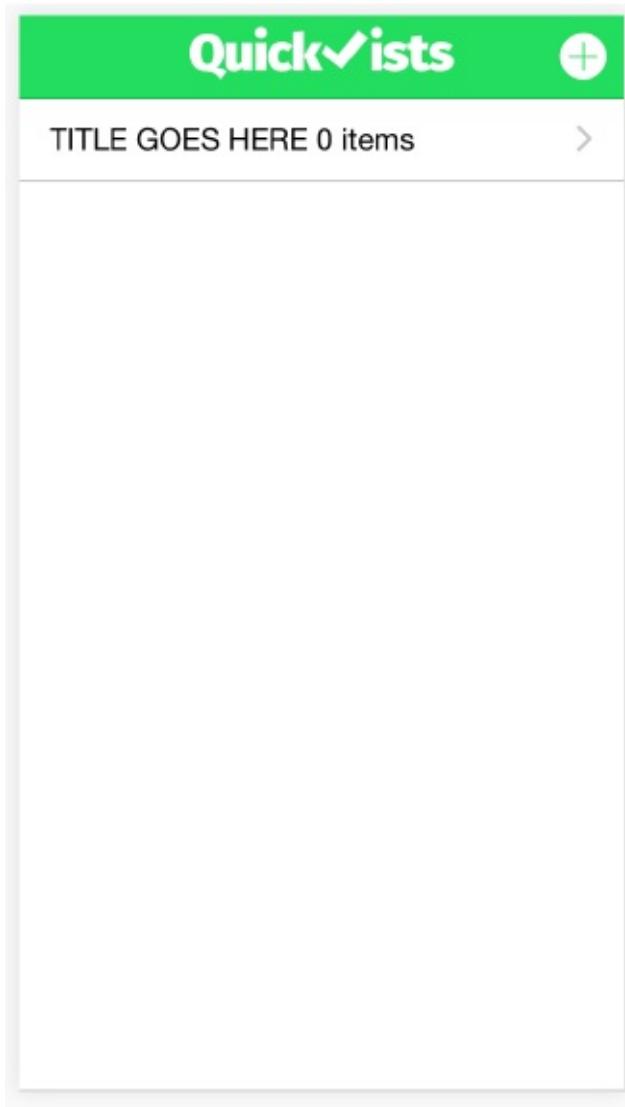
```
<something myName="myName">
```

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. 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:



I think we can both agree that's pretty ugly, but the structure is there. Please keep in mind that we haven't styled the logo properly yet, so depending on what size device you are looking at it with it may not fit as nicely as the image above does.

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 so far, but we are also going to need to handle the "logic" that happens in the application. 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 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 for that entity. In this case, we would use our service to create new checklists, to add items to the checklist, to update 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.

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 best to get pages to do as little work as possible. A page

should only contain "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 a button and some logic for that in the **home.page.ts** file. 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 the page.

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';

...

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.someFunction();
```

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
    }
  ]
};
```

```
        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: [
        {
            title: 'Item One',
            checked: false
        },
        {
            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/checklists.ts` to reflect the following:**

```
export interface Checklist {  
    id: string;  
    title: string;  
    items: ChecklistItem[];  
}  
  
export interface ChecklistItem {  
    title: string;  
    checked: boolean;  
}
```

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:

```
let myChecklist: Checklist = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      title: 'Item One',
      checked: false
    },
    {
      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 to save some time. 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.

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.

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-data.service.ts` to reflect the following:**

```
import { Injectable } from '@angular/core';
import { Checklist } from '../interfaces/checklists';

@Injectable({
```

```
    providedIn: 'root'
  })

export class ChecklistDataService {

  public checklists: Checklist[] = [];
  public loaded: boolean = false;

  constructor(){

  }

  load(): Promise<boolean> {
    return Promise.resolve(true);
  }

  createChecklist(data): void {

  }

  renameChecklist(checklist, data): void {

  }

  removeChecklist(checklist): void {

  }

  getChecklist(id): Checklist {
```

```
    return {
      id: '',
      title: '',
      items: []
    };
  }

addItem(checklistId, data): void {

}

removeItem(checklist, item): void {

}

renameItem(item, data): void {

}

toggleItem(item): void {

}

save(): void {

}

generateSlug(title): 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:

```
public 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. Notice that we have made this member variable `public`. We will be directly accessing the data contained in this service to display it in the pages in our 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 some of these functions are set to return a particular type of data, we are returning dummy data for any of the functions that aren't `void` (i.e. they are not supposed to return anything).

BEST PRACTICE ALERT! I feel the need to acknowledge a couple of things about this service that could be better. First of all, "checklists" and "checklist items" could be

considered two different entities, and so each should probably have their own service. The generateSlug function is also really just a generic function and not related specifically to checklists, so it would make sense to separate that out into some kind of "utilities" service or class. However, I've intentionally left these optimisations out for the sake of making this walkthrough easier to follow.

My general approach is to keep "best practice" in mind, but don't freak out over it. You can optimise things to the nth degree, and sometimes you don't get any real benefit out of it. Again, if you are working on a huge production application with a team of developers, you should definitely worry about this stuff. If you are building a small application by yourself... probably not as important. Especially as a beginner, if you get stuff like this wrong, or you aren't putting stuff in the "best" place, try not to worry too much.

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.

createChecklist

This function will be responsible for the creation of a new checklist.

> **Modify the createChecklist function to reflect the following:**

```
createChecklist(data): void {  
  
    this.checklists.push({  
        id: this.generateSlug(data.name),  
        title: data.name,  
        items: []  
    });  
  
    this.save();  
  
}  

```

It will accept some data as a parameter, and it will push a new checklist to the checklists array. Notice that we call the generateSlug function 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 sensible 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. Right now, this save call will do nothing.

renameChecklist

Next, we're going to define the `renameChecklist` function which, obviously, will allow us to rename a checklist.

> Modify the `renameChecklist` function to reflect the following:

```
renameChecklist(checklist, data): void {  
  
    let index = this.checklists.indexOf(checklist);  
  
    if(index > -1){  
        this.checklists[index].title = data.name;  
        this.save();  
    }  
  
}
```

This takes in a particular checklist as a parameter, as well as the data we want to update it with. We first find the particular checklist in our `checklists` array, and if it is found, we updated the title of that checklist with the data that was passed in.

removeChecklist

Next up we are going to add the ability to delete a checklist.

> Modify the `removeChecklist` function to reflect the following:

```
removeChecklist(checklist): void {
```

```
let index = this.checklists.indexOf(checklist);

if(index > -1){
    this.checklists.splice(index, 1);
    this.save();
}

}
```

This is quite similar to the `renameChecklist` function, except that we don't need to update it with any data. Instead of updating it, we just remove it from the array with `splice` and then trigger a save.

getChecklist

The `getChecklist` function is going to allow us to grab a particular checklist using its `id`.

> **Modify the `getChecklist` function to reflect the following:**

```
getChecklist(id): Checklist {
    return this.checklists.find(checklist => checklist.id ===
    id);
}
```

This function is a little different to the rest that we have added because this one actually returns something. This function will come in useful when we want to view the detail for a particular checklist. Earlier, we set up a route that would accept an `id` as a parameter. We can easily pass the `id` for a particular checklist through to the detail page, but we need *all* of the data for the checklist. This will allow us to easily get all of that data, whilst still only needing to pass the `id` through the URL. The `find` method we use here simply looks for a checklist with an `id` that matches the `id` passed into the `getChecklist`.

addItem

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. As I mentioned before, you may prefer to separate this out into its own service.

> **Modify the `addItem`, `removeItem`, and `renameItem` functions to reflect the following:**

```
addItem(checklistId, data): void {  
  
    this.getChecklist(checklistId).items.push({  
        title: data.name,  
        checked: false  
    });  
  
    this.save();
```

```
}

removeItem(checklist, item): void {

    let index = checklist.items.indexOf(item);

    if(index > -1){
        checklist.items.splice(index, 1);
        this.save()
    }

}

renameItem(item, data): void {

    item.title = data.name;
    this.save();

}
```

This is all basically the same idea as modifying our checklists.

toggleItem

Each of the items in our checklist can be toggled between a completed and incomplete state. This function simply flips the value of a particular item and then saves it to storage.

> Modify the `toggleItem` function to reflect the following:

```
toggleItem(item): void {
    item.checked = !item.checked;
    this.save();
}
```

generateSlug

Finally, we come to perhaps our most interesting function. Let's take a look.

> Modify the `generateSlug` function to reflect the following:

```
generateSlug(title): 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
    let exists = this.checklists.filter((checklist) => {
        return checklist.id.substring(0, slug.length) === slug;
    });
}
```

```
// If the title is already being used, add a number to make  
the slug unique  
  
if(exists.length > 0){  
    slug = slug + exists.length.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 number to the end of the `id` slug. We check for any `id` that has a matching `start`, so if there are multiple checklists with the same `id` they will be numbered in a sequence like this:

- my-checklist1
- my-checklist2
- my-checklist3

and so on. 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!

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 for a particular checklist so that data is displayed appropriately and we can interact with various functions like adding and deleting checklists.

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 relate directly to the template, any complex stuff 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 } from '@angular/core';
import { AlertController, IonList } from '@ionic/angular';
import { ChecklistDataService } from '../services/checklist-
data.service';

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

  @ViewChild(IonList) slidingList: IonList;

  constructor(public dataService: ChecklistDataService, private
  alertCtrl: AlertController) {

}

addChecklist(): void {

}
```

```
renameChecklist(checklist): void {  
  
}  
  
removeChecklist(checklist): void{  
  
}  
  
}
```

The first 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 `ChecklistDataService` into the constructor to make it available throughout the class.

We are also importing `IonList` and `ViewChild`. We can use `@ViewChild` as we do above the constructor to grab a reference to an element 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.

Notice that we use `public` for the checklist service and `private` for the alert controller?

That is because we will only be accessing the alert controller from within this class, but we will be accessing the checklist service directly from our template so it needs to be `public`.

Let's implement the functions now.

addChecklist

> **Modify the `addChecklist` function to reflect the following:**

```
addChecklist(): void {  
  
  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: data =>  
          this.checklistService.addChecklist(data.name);  
      }  
    ]  
  }).present();  
}
```

```
    handler: (data) => {
      this.dataService.createChecklist(data);
    }

  }
]

}).then((prompt) => {
  prompt.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.

The `create` method will return a promise that resolves once the alert has been created. This is why we chain a `then` onto this, which will be triggered when the promise resolves. We then take the alert that was created, and we call the `present` method to display it to the user.

renameChecklist

> Modify the `renameChecklist` function to reflect the following:

```
renameChecklist(checklist): void {  
  
  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: (data) => {  
          this.dataService.renameChecklist(checklist, data);  
        }  
      }  
    ]  
  }  
}
```

```
}).then((prompt) => {
    prompt.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 `renameChecklist` method in our checklist service.

removeChecklist

> **Modify the `removeChecklist` function to reflect the following:**

```
removeChecklist(checklist): void{
    this.slidingList.closeSlidingItems().then(() => {
        this.dataService.removeChecklist(checklist);
    });
}
```

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.

In a way, the function in our page is kind of just acting as a proxy for the checklist service we created. In this case, since we aren't doing anything else besides calling a function in the checklist service, you could even just skip having this function completely and instead call `dataService.removeChecklist(checklist)` directly from your template.

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

<ion-content>

  <ion-list lines="none">
    <ion-item-sliding *ngFor="let checklist of
```

```

dataService.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" expandable
(click)="renameChecklist(checklist)"><ion-icon slot="icon-only"
name="clipboard"></ion-icon></ion-item-option>
        <ion-item-option color="danger" expandable
(click)="removeChecklist(checklist)"><ion-icon slot="icon-only"
name="trash"></ion-icon></ion-item-option>
    </ion-item-options>

    </ion-item-sliding>

</ion-list>
</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
dataService.checklists">

  <ion-item [routerLink]="'/checklists/' + checklist.id"
routerDirection="forward">
    <ion-label>{{checklist.title}}</ion-label>
    <span>{{checklist.items.length}} items</span>
  </ion-item>

  <!-- sliding 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 available in our `checklists` service. 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 have the square brackets around `routerLink` because we want the path

to be /checklists/ plus the checklist.id - by using square brackets around the property, this expression will be calculated before assigning the value to routerLink. 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 } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { AlertController, IonList } from '@ionic/angular';
import { ChecklistDataService } from '../services/checklist-
data.service';
import { Checklist } from '../interfaces/checklists';
```

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

  @ViewChild(IonList) slidingList: IonList;

  private slug: string;
  public checklist: Checklist;

  constructor(private alertCtrl: AlertController, private route: ActivatedRoute, private dataService: ChecklistDataService){

  }

  ngOnInit(){

  }

  loadChecklist(){

  }

  addItem(): void {

  }
}
```

```
removeItem(item): void {  
  
}  
  
renameItem(item): void {  
  
}  
  
toggleItem(item): void {  
  
}  
  
}
```

Some of these functions will look very similar to the functions 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. Let's get into defining the functions 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` function to reflect the following:

```
ngOnInit(){
    this.slug = this.route.snapshot.paramMap.get('id');
    this.loadChecklist();
}
```

We use this function 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`. In the next function, we will use that `id` or `slug` to load the data for that checklist

loadChecklist

> Modify the `loadChecklist` function to reflect the following:

```
loadChecklist(){
    if(this.dataService.loaded){
        this.checklist = this.dataService.getChecklist(this.slug);
    } else {
        this.dataService.load().then(() => {
            this.checklist =
        this.dataService.getChecklist(this.slug);
    });
}
```

```
    }  
}  
}
```

All this function is responsible for is calling the `getChecklist` function from our checklist service and supplying it with the `id` of the checklist that we are trying to do. It will then assign the result to the class member `this.checklist`. However, we need to make sure the data has finished loading before we try to access it, so we call the `load` method and trigger our code inside of the success handler for the promise (which will only trigger once the data has finished loading). So that we don't need to load from storage every single time, we check if the `loaded` flag has been set in the service before triggering the `load`.

addItem, removeItem, and renameItem

I have bunched these functions 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` functions to reflect the following:

```
addItem(): void {  
  
    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.dataService.addItem(this.checklist.id, data);
    }
  }
]
}).then((prompt) => {
  prompt.present();
});
}

removeItem(item): void {
  this.slidingList.closeSlidingItems().then(() => {
    this.dataService.removeItem(this.checklist, item);
  });
}

```

```
}

renameItem(item): void {

    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.dataService.renameItem(item, data);
                }
            }
        ]
    }).then((prompt) => {
        prompt.present();
    })
}
```

```
});
```

```
}
```

toggleItem

> **Modify the toggleItem function to reflect the following:**

```
toggleItem(item): void {  
  this.dataService.toggleItem(item);  
}
```

This function just has the responsibility of calling the toggleItem function in the checklist service. Just like before, this function isn't strictly required and you could just call it directly from the template if you wanted to.

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>  
  <ion-toolbar color="success">
```

```

<ion-title>
  {{ checklist?.title }}
</ion-title>
<ion-buttons slot="start">
  <ion-back-button defaultHref="/checklists"></ion-back-
button>
</ion-buttons>
<ion-buttons slot="end">
  <ion-button (click)="addItem()"><ion-icon slot="icon-only"
name="add-circle"></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>
      <ion-label>{{ item.title }}</ion-label>
      <ion-checkbox [checked]="item.checked"
        (ionChange)="toggleItem(item)"></ion-checkbox>
    </ion-item>

    <ion-item-options side="end">
      <ion-item-option tappable color="light">

```

```
(click)="renameItem(item)"><ion-icon slot="icon-only"
name="clipboard"></ion-icon></ion-item-option>
<ion-item-option tappable 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.

However, there is one important difference. Notice that we are using syntax like this:

```
{{ checklist?.title }}
```

```
let item of checklist?.items
```

This question mark is the **safe navigation operator**. Basically, it allows you to reference

properties of an object even if the object does not exist without causing an error. The issue here is that the template might try to access the checklist data before it has loaded in - in that case, it would cause an error. If we try to access `checklist.title`, and `checklist` is currently `undefined` then we are going to get an error because of course `undefined` does not have a property called `title`. By using the safe navigation operator, it won't trigger an error. Once the data finishes loading in, the template will update with the correct data.

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.

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 function every time some data changes, we just need to implement that function. 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. In order to make use of Ionic's Storage API, we will need to import it into our service and inject it through the constructor.

> Add the following import to **src/app/services/checklist-data.service.ts**:

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

> Modify the constructor in **src/app/services/checklist-data.service.ts** to reflect the following:

```
constructor(private storage: Storage){  
}
```

With the storage service added to our own service, we can now implement our save and load functions.

> Modify the save function in `src/app/services/checklist-data.service.ts` to reflect the following:

```
save(): void {
  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 function in `src/app/services/checklist-data.service.ts` to reflect the following:

```
load(): Promise<boolean> {
  return new Promise((resolve) => {
    this.storage.get('checklists').then((checklists) => {
      if(checklists != null){
        this.checklists = checklists;
      }
    })
  })
}
```

```
this.loaded = true;
resolve(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, and so we set up a .then handler to deal with the data once it is ready. Once we do get the data, we check to see if it is not empty, and then we update our this.checklists member variable with the data.

We also wrap all of this inside of our own Promise. We want to load the data into the service, but in our checklist page we are also relying on being able to tell *when* the data has finished loading. So, we have our load function return its own promise that resolves after the call to storage resolves. This ensures that the data is ready to be accessed when we make our call from the checklist page to load a specific checklist. It is worth noting that there are better ways to design this, using Observables/BehaviorSubject for example, but this is a beginner friendly way to go about it.

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 inside of the constructor inside of our root component. Our root component is the first component added to the application, and `ngOnInit` will trigger whenever a component is initialised. So, this will mean our code gets triggered right when the application starts up.

> **Modify `src/app/app.component.ts` to reflect the following:**

```
import { Component } from '@angular/core';
import { Plugins } from '@capacitor/core';
import { ChecklistDataService } from './services/checklist-
data.service';

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {

  constructor(private DataService: ChecklistDataService) {

    this.dataService.load();
  }
}
```

```
SplashScreen.hide().catch((err) => {
  console.warn(err);
});

StatusBar.hide().catch((err) => {
  console.warn(err);
});

}

}
```

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 Theming

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 on the last slide 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>
        
    </ion-slide>

    <ion-slide>
        
    </ion-slide>

    <ion-slide>
        
    </ion-slide>

    <ion-slide>
        <ion-grid>
            <ion-row>
                <ion-col>
                    <ion-button color="light" routerLink="/" routerDirection="forward" style="margin-top:20px;">Start Using Quicklists</ion-button>
                </ion-col>
            </ion-row>
            <ion-row>
                <ion-col>
                    
                </ion-col>
```

```
</ion-row>  
</ion-grid>  
</ion-slide>  
  
</ion-slides>  
  
</ion-content>
```

The first thing you might notice about this is that it doesn't contain a header or toolbar, only the content area. 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.

So 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 which contains a button to go to the home page.

The last slide is a little more complicated because we are making use of `<ion-row>` and `<ion-col>` so that we can position the button where we want it. These two directives make up Ionic's grid system, where rows get placed underneath one another, and cols within those rows appear side by side. This diagram should help illustrate that:



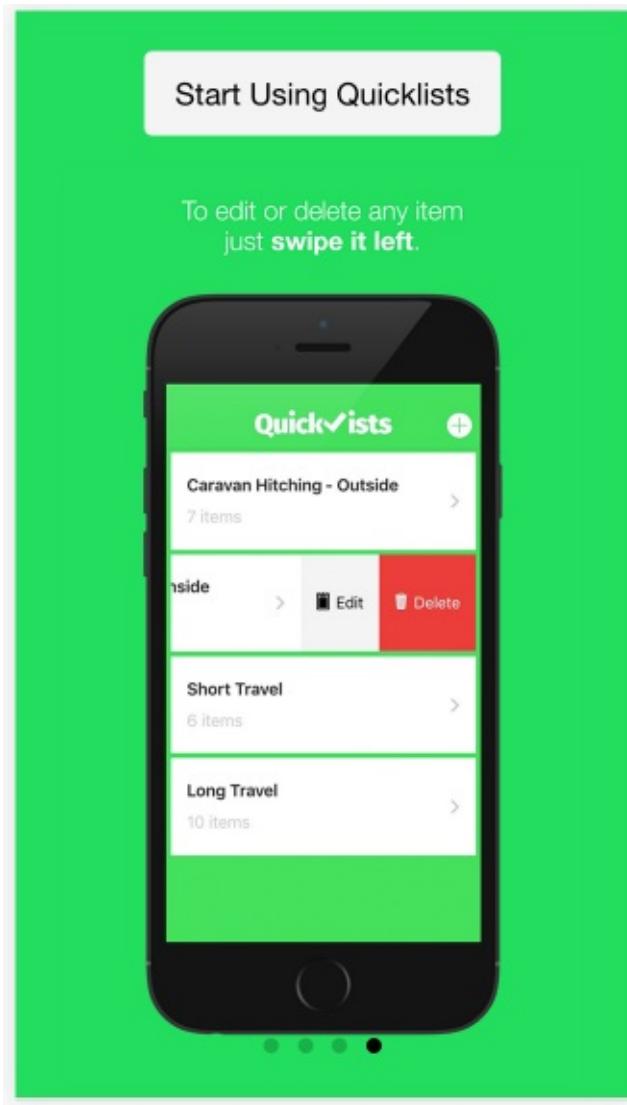
= **row**

= **column**

This is a very simple example because we just want the button to appear above the image, but you can create quite complex layouts by supplying a width to the cols like this:

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

The first column would take up 2 columns, and the second would take up 6 (based on a total page width of 12 columns). You can also nest rows and columns as much as you like. The result of our grid layout will look like this:



We don't even need to define any functionality in the class for the Intro page, 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 going to set up the `ngOnInit` lifecycle hook for the home page, and we are also going to

import NavController from the Ionic library so we can force the user back to the intro page if necessary.

> Modify the @ionic/angular import in src/app/home/home.page.ts to reflect the following:

```
import { AlertController, IonList, NavController } from  
'@ionic/angular';
```

> Modify the @angular/core import in src/app/home/home.page.ts to reflect the following:

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

> Add an @ionic/storage import in src/app/home/home.page.ts:

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

> Change to class to implement OnInit in src/app/home/home.page.ts:

```
export class HomePage implements OnInit {
```

> **Modify the constructor in src/app/home/home.page.ts to reflect the following:**

```
constructor(  
  public dataService: ChecklistDataService,  
  private alertCtrl: AlertController,  
  private storage: Storage,  
  private navCtrl: NavController) {  
  
}
```

> **Add an ngOnInit function beneath the constructor in src/app/home/home.page.ts:**

```
ngOnInit(){  
  
  this.storage.get('introShown').then((result) => {  
  
    if(result == null){  
      this.storage.set('introShown', true);  
      this.navCtrl.navigateRoot('/intro');  
    }  
  })  
}
```

```
}
```

```
});
```

```
}
```

We're importing and making use of the Storage service again now, and we have also imported the NavController and injected it into our constructor. Previously, we have just been navigating between pages using an routerLink, but in this case, we want to change the route programatically. 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.

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 debugger by going to **Application > Clear storage.**

Theming

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 CSS4 variables.

Keep in mind that since we use SASS for CSS in Ionic, we are able to nest our CSS selectors styles.

> **Modify src/intro/intro.page.scss to reflect the following:**

```
ion-slide {  
    background-color: #32db64;  
}  
  
ion-slide img {  
    height: 85vh !important;  
    width: auto !important;  
}
```

These styles will make the background colour of the slides green, and also set the images inside of the slides to take up 85% of the available viewport height. Since we have added these styles specifically to the .scss file for the IntroPage, they will only apply to the

<app-intro> component.

Let's move on to the home page now.

> **Modify src/home/home.page.scss to reflect the following:**

```
ion-item-sliding {  
  margin: 5px;  
}  
  
ion-label {  
  font-size: 1.2em;  
  font-weight: bold;  
  color: #282828;  
  padding-top: 10px;  
  padding-bottom: 10px;  
}  
  
ion-item span {  
  font-weight: 400;  
  margin-right: 10px;  
}
```

We're not doing anything too crazy here, just adding a few tweaks to the margins, padding and colours. Now let's do the same to the checklist page.

> Modify `src/checklist/checklist.page.scss` to reflect the following:

```
ion-item-sliding {  
  margin: 5px;  
}  
  
ion-checkbox {  
  font-size: 0.9em;  
  font-weight: bold;  
  color: #282828;  
  padding-top: 0px;  
  padding-bottom: 0px;  
  padding-left: 4px;  
  border: none !important;  
}  
  
ion-item-content {  
  border: none !important;  
}  
  
ion-checkbox {  
  border-bottom: none !important;  
}
```

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 `src/global.scss`:

```
ion-title img {  
  max-height: 30px;  
  width: auto;  
}
```

Here we're just setting a maximum height for the logo. Finally, we are going to mess with some CSS4 variables.

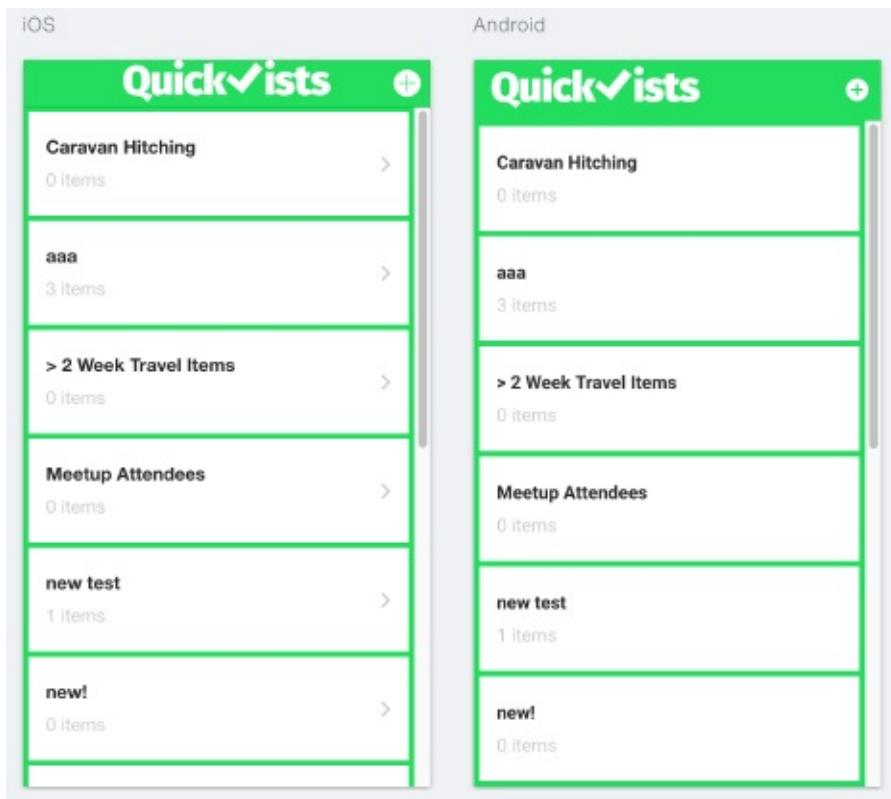
> Add the following above the existing `:root` selector in `src/theme/variables.scss`:

```
ion-content {  
  
  --ion-background-color: #32db64;  
  
  ion-item-sliding {  
    --ion-background-color: #fff;  
  }  
  
}
```

You will see a bunch of Ionic's own CSS variables that you can change in this file if you want, but we can also configure any CSS4 variables we like in our own selectors, or even our own `:root` selector that will apply the change application wide. In this case, we are changing the background colour of everything inside `<ion-content>` to green. We are also setting the background colour for sliding items to white. We don't want to make these variable changes to *everything* (e.g. by putting it inside of `:root`) because that would also make the backgrounds for our inputs green, and it would give any items across the whole application a white background. We are specifically changing the CSS4 variables *only* inside of the `ion-content` and `ion-item-sliding` selectors - anything outside of those selectors will not be affected.

NOTE: Make sure that you do not delete the Ionic CSS Variables in the **variables.scss** file.

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.

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 walkthrough. 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 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:

- Retheme the application with your own styling, try different colours, padding, margins and so on **[EASY]**
- Add a 'Date Created' 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 notes to any specific checklist item **[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.

Giflist

Lesson 1: Giflist Introduction

Giflist was the first application I created with Ionic 2, and it was originally built on one of the really early alpha versions of Ionic 2. It has been updated a bit since then, and more recently with Ionic 4 and Capacitor for this version of the book. Even though it was the first app I built it's still my favourite, so I'm excited to walk you through building it.

I think it's my favourite because it's just a really fun 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 funny GIFs?

About Giflist

Giflist is a pretty simple application, 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](#)

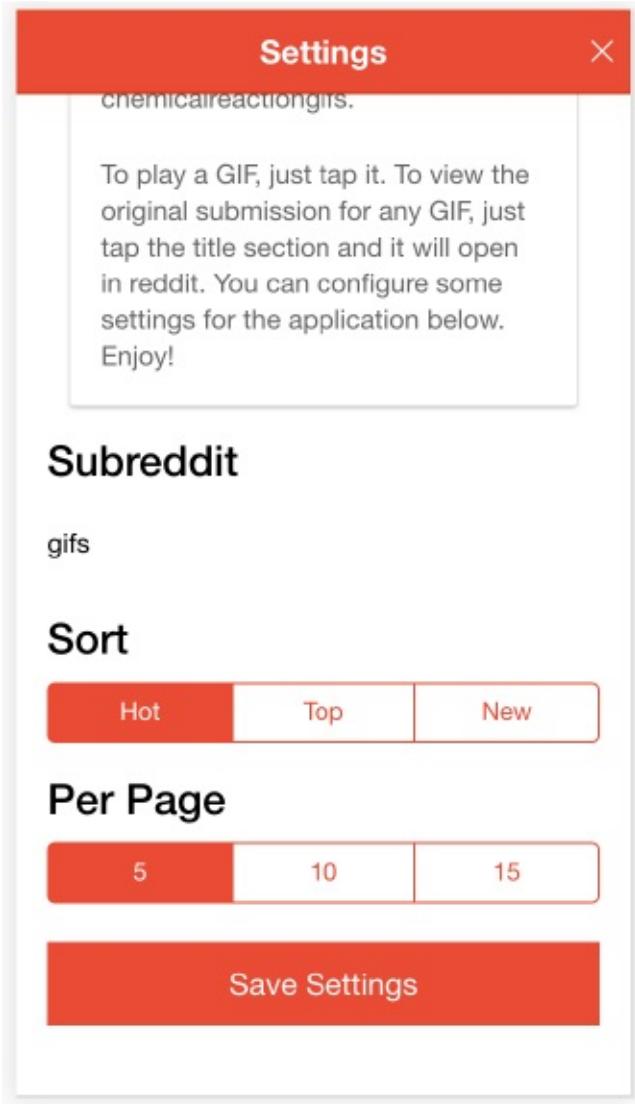
As well as just fetching GIFs from reddit, users will also be able to supply some settings to configure the application to their preferences. Although the concept is pretty simple, there are a few interesting things that you will learn through building it, this will include:

- Fetching data from a 3rd party API
- Data storage
- Using Observables
- Theming
- 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 set the following options: *default subreddit, sort order, GIFs per page*
- The users settings will be remembered upon returning to the application

and a couple of screenshots to put it into context:





Lesson Structure

1. [Getting Ready](#)
2. [The List Page](#)
3. [The Reddit API and HTML5 Video](#)
4. [Integrating the Data Service](#)
5. [Settings](#)
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
```

or

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

before you continue. If you run into any trouble installing Ionic or generating new projects, make sure that you have the most recent [NodeJS LTS](#) version installed. After you have that installed, you should also run the following command:

```
npm uninstall -g ionic
```

before attempting to install again.

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 giflist blank --type=angular
```

When asked, **do not** integrate the Ionic Appflow SDK.

> **Make the new project your current working directory by running the following command:**

```
cd giflist
```

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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

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 **home** 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 data service to handle storing and retrieving the user's settings, and a service to help us retrieve data from Reddit.

> Run the following command to generate a Data service:

```
ionic g service services/Data
```

NOTE: Notice that we use services/Data instead of just Data 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 a Reddit service:

```
ionic g service services/Reddit
```

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 { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', loadChildren:
    './home/home.module#HomePageModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  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

Capacitor is not included by default in our applications, so we need to set it up. You should still complete this step even if you don't intend to build for iOS and Android because we will also be using Capacitor for the web platform.

To set up Capacitor in an Ionic application, all you need to do is run a single command.

> Run the following command to enable Capacitor:

```
ionic integrations enable capacitor
```

This will handle setting up the Capacitor CLI and Capacitor Library in your project.

Keep in mind that this will give your application a default 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 now. So, you may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`.

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. 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
```

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your application code and dependencies over 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.

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'
})
export class AppComponent {

  constructor() {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });
  }
}
```

```
}
```

All we are doing here is hiding the StatusBar but you can use this API to style the Status Bar however you like. 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 likely to get wiped by the operating system).

To enable Ionic storage, you need to install it.

> **Install Ionic Storage with the following command:**

```
npm install @ionic/storage --save
```

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 { RouterModule, RouteReuseStrategy, Routes } 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 { RouterModule, RouteReuseStrategy, Routes } 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 and entryComponents.

> **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 { ReactiveFormsModule, FormsModule } from
'@angular/forms';
import { RouterModule } from '@angular/router';
```

```
import { HomePage } from './home.page';
import { SettingsPage } from '../settings/settings.page';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    IonicModule,
    RouterModule.forChild([
      {
        path: '',
        component: HomePage
      }
    ])
  ,
  declarations: [HomePage, SettingsPage],
  entryComponents: [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 --save
```

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 even 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 { DataService } from './data.service';

import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
```

```
)  
export class RedditService {  
  
    public settings = {  
        perPage: 10,  
        subreddit: 'gifs',  
        sort: '/hot'  
    };  
  
    public posts: any[] = [];  
  
    public loading: boolean = false;  
    private page: number = 1;  
    private after: string;  
    private moreCount: number = 0;  
  
    constructor(private http: HttpClient, private dataService:  
DataService) {  
  
    }  
  
    load(): void {  
  
    }  
  
    fetchData(): void {  
  
    }  
}
```

```
nextPage(): void {  
  
}  
  
resetPosts(){  
  
}  
  
changeSubreddit(subreddit): void {  
  
}  
  
}
```

This is the basic skeleton for the service. It defines a bunch of member variables and a `fetchData` function that will handle grabbing the data from Reddit (this will be the most significant function in the service). What the member variables do may not be immediately obvious, so let's go through them (in order):

- We have a `settings` object that will allow the user to define specifically how the posts should be retrieved from Reddit. We also supply some default settings to be used for this object.
- The `posts` array will hold the gifs that we pull in from Reddit
- The `loading` boolean will keep track of whether we are currently attempting to load in data from Reddit or not

- The page 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".
- The after string keeps track 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".
- The moreCount 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 the service to keep trying to fetch data if it doesn't get enough before giving up.

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 data service (which we are yet to implement). We are also importing the map operator from RxJS, but we are going to talk about that a little later.

We will finish up this service later, but for now, let's move on to the home page layout.

The Layout

Before we start building the layout for our home page, let's take a look at what we're building:



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 'Load More' button which sits at the bottom of the list when the user taps this it will load in the next page of GIFS.

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="secondary">

    <ion-title>
      <ion-searchbar placeholder="enter subreddit name...">
    </ion-searchbar>
    </ion-title>

    <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 no-lines>

    <div tappable>
      <ion-item>
        GIF GOES HERE
      </ion-item>
    <ion-list-header>
```

```

<ion-label>TITLE GOES HERE</ion-label>
</ion-list-header>
</div>

<ion-item *ngIf="redditService.loading" no-lines style="text-align: center;">
  <ion-spinner name="lines-small"></ion-spinner>
</ion-item>

</ion-list>

<ion-button color="light" expand="full"
(click)="loadMore()">Load More...</ion-button>

</ion-content>

```

In the first part of this code we are setting up the header:

```

<ion-header>
  <ion-toolbar color="secondary">

    <ion-title>
      <ion-searchbar placeholder="enter subreddit name...">
    </ion-searchbar>
  </ion-title>

```

```
<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 secondary color attribute to the `<ion-toolbar>` which will alter its style to use the secondary colour, which we will be changing later.

We're using `<ion-title>`, which is usually used to supply a title to be displayed on the toolbar, to position our search bar in the middle. Usually, you'd have a separate toolbar for a searchbar so that it can take up the whole space, but I didn't want to clutter the screen too much so we're going to bend the rules a bit here with this. In order for it to display properly, we'll need to add a few custom styles later.

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` function 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 no-lines>

  <div tappable>
    <ion-item>
      GIF GOES HERE
    </ion-item>
    <ion-list-header>
      <ion-label>TITLE GOES HERE</ion-label>
    </ion-list-header>
  </div>

  <ion-item *ngIf="redditService.loading" no-lines style="text-align: center;">
    <ion-spinner name="lines-small"></ion-spinner>
  </ion-item>

</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 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 the `no-lines` attribute 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`.

As well as our GIF items, we are going to add one additional item at the bottom of the list which will contain a loading animation. This will be used to display a spinning animation when new GIFs are being fetched, but since we only want it to display when loading is occurring we use the `*ngIf` directive to control when it displays. In this case, the loading animation will only display when the loading boolean in our Reddit service evaluates to be true.

The last bit of code we have in our template is the load more button:

```
<ion-button color="light" expand="full" (click)="loadMore()">Load  
More...</ion-button>
```

Nothing too crazy is happening here, we supply the `light` color attribute to change the colour and we have a `(click)` listener set up that will eventually call the `loadMore()` function we will define in our class definition for the home page. Also, notice that we use `expand="full"` to style the button to take up the full width available to it.

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 the code for the entire class first and then we will discuss it bit by bit.

> Modify src/app/home/home.ts to reflect the following:

```
import { Component, OnInit } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { DataService } from '../services/data.service';
import { RedditService } from '../services/reddit.service';
import { FormControl, FormGroup } from '@angular/forms';
import { Plugins } from '@capacitor/core';

import { SettingsPage } from '../settings/settings.page';

import { debounceTime } from 'rxjs/operators';
import { distinctUntilChanged } from 'rxjs/operators';

const { Browser, Keyboard } = Plugins;

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

```
export class HomePage implements OnInit {

    public subredditForm: FormGroup;

    constructor(private dataService: DataService, public redditService: RedditService, private modalCtrl: ModalController) {
        this.subredditForm = new FormGroup({
            subredditControl: new FormControl('')
        });
    }

    ngOnInit(){
    }

    showComments(post): void {
    }

    openSettings(): void {
    }

    playVideo(e, post): void {
    }
}
```

```
}

loadMore(): 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. So, let's start talking through it.

First up are our **import** statements:

```
import { Component, OnInit } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { DataService } from '../../../../../services/data.service';
import { RedditService } from '../../../../../services/reddit.service';
import { FormControl, FormGroup } from '@angular/forms';
import { Plugins } from '@capacitor/core';

import { SettingsPage } from '../settings/settings.page';

import { debounceTime } from 'rxjs/operators';
import { distinctUntilChanged } from 'rxjs/operators';
```

```
const { Browser, Keyboard } = 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 Modal Controller, and we also import both of our own services (the data service and the Reddit service).

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 also set up some references for the Capacitor plugins that we will be making use of in this page.

In the next section of code, we have our constructor function and `ngOnInit` 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 too much "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).

```
public subredditForm: FormGroup;

constructor(private dataService: DataService, public redditService: RedditService, private modalCtrl: ModalController)
{

  this.subredditForm = new FormGroup({
    subredditControl: new FormControl('')
  });

}

ngOnInit(){
```

Above the constructor we set up a single 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 reread the **User Input** lesson from the basics section of the book.

Let's take a look at the remaining code in the class:

```
showComments(post): void {  
}  
  
openSettings(): void {  
}  
  
playVideo(e, post): void {  
}  
  
loadMore(): void {  
}
```

The rest of these functions 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). So you rarely have to create the Observables yourself. But, there's a lot of fancy stuff you can do with them if you want to.

We are using the **FormControl** service we imported before to create a "FormControl" that will supply our Observable. FormControls 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:**

```
<ion-title>
  <form [formGroup]="subredditForm">
    <ion-searchbar placeholder="enter subreddit name...">
      <input formControlName="subredditControl" value="" />
    </ion-searchbar>
  </form>
</ion-title>
```

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.load();

    this.subredditForm.get('subredditControl').valueChanges.pipe(
        debounceTime(1500),
        distinctUntilChanged()
    ).subscribe((subreddit: any) => {

        if(subreddit.length > 0){
            this.redditService.changeSubreddit(subreddit);
            Keyboard.hide().catch((err) => {
                console.warn(err);
            });
        }
    })
}
```

```
    })
```

```
}
```

First, we call the `load` function in the Reddit service. 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(  
  debounceTime(1500)  
) .subscribe
```

We would only run the code when the input has changed, and when there hasn't been another change within 1.5 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 1.5 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 `changeSubreddit` function. We also call the `hide()` method on the `Keyboard` plugin; since we are kind of going against the default behaviour of an input field (which would usually be submitted) the keyboard doesn't know when to close, so we trigger that manually after the operation has finished.

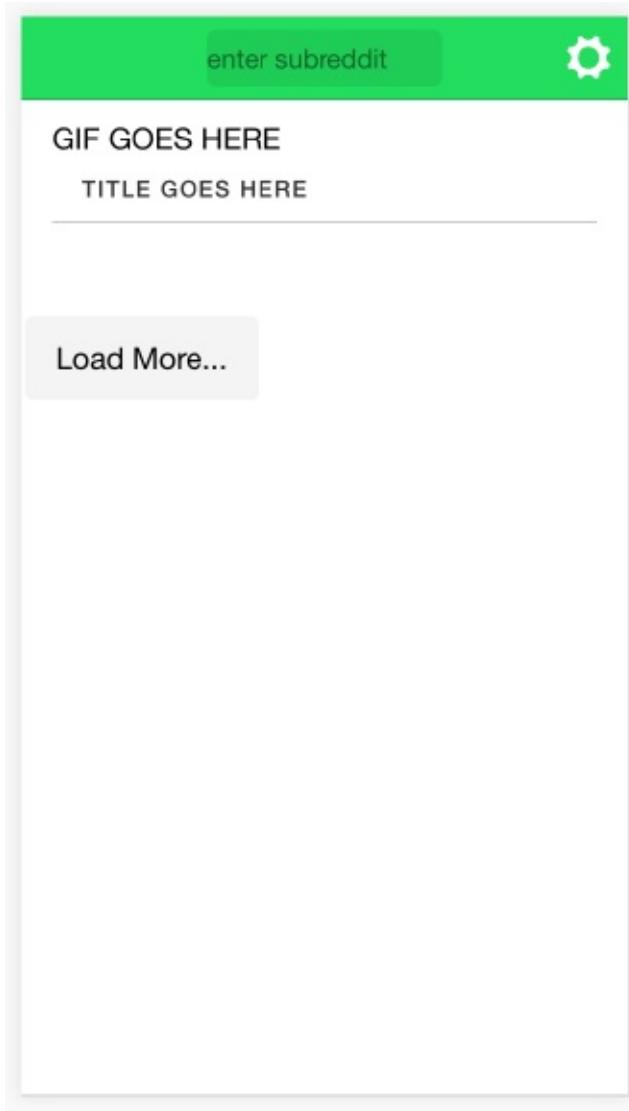
This will probably be one of the most confusing parts of this application, especially if you're completely new to Observables. As you can tell this has allowed us to create some pretty useful functionality really easily, but we could have just as easily used the normal `ngModel` approach and just used a button for triggering searches instead. The downside to this is that we miss all the fancy observable behaviour.

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:



Pretty ugly right? And you're even going to get some errors in the console too. 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.

Lesson 4: The Reddit API and HTML5 Video

One of the most interesting things about **Giflist** 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.

So what we are going to do is make sure we only ever pull in GIFs that provide a **.webm** or **.gifv** format. This means we won't be displaying GIFs at all, we will be displaying **videos**.

We're going to display those videos using the HTML5 `<video>` tag. It's important to remember when building HTML5 mobile applications, we can use *any* HTML5 feature 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 plain old HTML5 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 a few important issues with HTML5 video.

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 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 both inline and in fullscreen
- 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 and whether it plays inline can all be controlled

One of the most important things to know though is that the behaviour of the `<video>` element is different depending on which platform the application is running on.

- On iOS videos play in fullscreen by default, but can be forced to play inline by using the **webkit-playsinline** attribute. However, this is not universal across all iOS devices. On smaller devices, fullscreen will still be forced even if you specify the **webkit-playsinline** attribute (basically, it's not possible to change this)
- On Android videos play inline by default but can be made fullscreen

Given those behavioural differences, we then need to figure out how we want to approach the problem. We basically have two options:

1. Accept the default behaviour and maintain the same code for both platforms
2. Check for the platform we are running on and run different code to achieve the desired behaviour.

Personally, I think playing the videos inline achieves a more desirable result. But since it's not possible to achieve that on small iOS devices, I decided to just go with the default behaviour. This means that there will be a difference in behaviour between iOS and

Android, but I think both solutions are perfectly acceptable and it will keep our code base nice and simple.

So, let's get to work. We're going to finish implementing a few more of the functions we require to get our application working.

Fetching Data from Reddit

We're going to start off with the most interesting and complicated function in our Reddit service, which is pretty much the core feature of the entire application: `fetchData()`. We will add the code first and then walk through it

> **Modify the `fetchData` function in `src/app/services/reddit.service.ts` to reflect the following:**

```
fetchData(): 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.settings.sort + '/.json?  
        limit=100';  
  
    //Only grab posts from the API after the last post we  
    //retrieved (if we've already fetched some data)  
    if(this.after){
```

```
url += '&after=' + this.after;
}

this.loading = true;

this.http.get(url).pipe(
    //Modify the response to return data in a more friendly
format
    map((res: any) => {
        console.log(res);

        let response = res.data.children;
        let validPosts = 0;

        //Remove any posts that don't provide a GIF in a suitable
format
        response = response.filter((post) => {
            //If we've already retrieved enough posts for a page,
            we don't want anymore
            if(validPosts >= this.settings.perPage){
                return false;
            }

            //We only want to keep .gifv and .webm formats, and
            then convert them to mp4
        })
    })
)
```

```

        if(post.data.url.indexOf('.gifv') > -1 ||

post.data.url.indexOf('.webm') > -1){

    post.data.url = post.data.url.replace('.gifv',
'.mp4');

    post.data.url = post.data.url.replace('.webm',
'.mp4');

    //If a preview image is available, assign it to the
post as 'snapshot'

    if(typeof(post.data.preview) != "undefined"){

        post.data.snapshot =
post.data.preview.images[0].source.url.replace(/\&/g, '&');

        //If the snapshot is undefined, change it to be
blank so it doesnt use a broken image

        if(post.data.snapshot == "undefined"){

            post.data.snapshot = "";

        }

    }

    else {

        post.data.snapshot = "";

    }

}

validPosts++

```

```

        return true;

    } else {
        return false;
    }

});

//If we had enough valid posts, set that as the "after",
otherwise just set the last post
if(validPosts >= this.settings.perPage){
    this.after = response[this.settings.perPage -
1].data.name
} else if(res.data.children.length > 0) {
    this.after = res.data.children[res.data.children.length -
1].data.name;
    console.log(this.after);
}

return response;
}

).subscribe((data: any) => {

    console.log(data);

    //Add new posts we just pulled in to the existing posts
}

```

```
this.posts.push(...data);

//Keep fetching more GIFs if we didn't retrieve enough to
fill a page

//But give up after 20 tries if we still don't have enough
if(this.moreCount > 50){

    console.log("giving up");

    //Time to give up!
    this.moreCount = 0;
    this.loading = false;

} else {

    //If we don't have enough valid posts to fill a page, try
    fetching more data
    if(this.posts.length < (this.settings.perPage *
this.page)){
        this.fetchData();
        this.moreCount++;
    } else {
        this.loading = false;
        this.moreCount = 0;
    }
}
```

```
}, (err) => {
    console.log(err);
    //Fail silently, in this case the loading spinner will just
    continue to display
    console.log("Can't find data!");
};

}
```

That's a pretty big function. 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.

First, we build the URL that we are going to use to fetch data from the Reddit API. We use whatever **subreddit** and **sort** values the user currently has set. You can modify this URL with any values for these that you like and it will spit out the relevant JSON. If we have an **after** value set then we supply that as well. This is how the Reddit API does "pagination", if the user has hit the "Load More" button three times then we only want to return the posts for the third page, e.g results 10-15 if the user has a page size of 5. With the Reddit API, you can supply a posts "name" as the after parameter and it will return only the posts after that post.

Then we use that URL to make an HTTP request and subscribe to the Observable returned, but before doing that we use pipe to modify the observable with the map operator. This is the same idea as modifying the observable for the form control that our searchbar uses, except that we are modifying the response coming back from our HTTP request.

Inside of this map function, we loop through each of the posts returned from the API to filter out any that we don't want. When we are looping through the newly loaded posts we are doing a few things:

- Removing any posts that are not in the .gifv or .webm formats
- Converting .gifv and .webm links to .mp4
- Assigning a preview image to be used as the video poster if it is available

Once we have finished the loop we should have an array of posts in the format we need, ready to be displayed in the list. There's one more important issue we need to take care of, though. If we are loading in say 10 posts per page from the Reddit API, but then only 3 of those are suitable GIFS, our page size is only going to be 3. This might not fly too well with the user.

To solve this issue, we will recursively fetch more data by calling the `fetchData()` function from within the `fetchData()` function. This will keep adding more and more posts to the posts array until we have our full 10 (or whatever the page size currently is) GIFs. We need to set a limit though because this function could just run infinitely, so if we still don't have enough GIFs after 50 tries then we give up.

Also, notice that we have an error handler for the `http.get` request as well. If a successful request is made, all of the code we just discussed will run, but if not (i.e. if the subreddit the user is trying to retrieve results in a 404) then the error will be passed into the error handler and that will run instead.

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 src/app/home/home.page.html to reflect the following

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

    <ion-title>
      <form [formGroup]=" subredditForm ">
        <ion-searchbar placeholder="enter subreddit name..." formControlName=" subredditControl " value=""></ion-searchbar>
      </form>
    </ion-title>

    <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 no-lines>

    <div tappable (click)=" playVideo ($event, post) " *ngFor="let post of redditService.posts " >
      <ion-item>
```

```

        <video loop [src]="post.data.url"
[poster]="post.data.snapshot">
    </video>
</ion-item>
<ion-list-header (click)="showComments(post)" style="text-align: left;">
    <ion-label>{{post.data.title}}</ion-label>
</ion-list-header>
</div>

<ion-item *ngIf="redditService.loading" no-lines style="text-align: center;">
    <ion-spinner name="lines-small"></ion-spinner>
</ion-item>

</ion-list>

<ion-button color="light" expand="full" (click)="loadMore()" *ngIf="redditService.posts.length">Load More...</ion-button>

</ion-content>

```

We are now using an *ngFor to loop over every post that has been loaded into our Reddit service.

As you can see above we are now using the URL of the post as the source of the video element, as well as supplying the preview snapshot as the poster and the title in the

header. We've also added a few more things here as well.

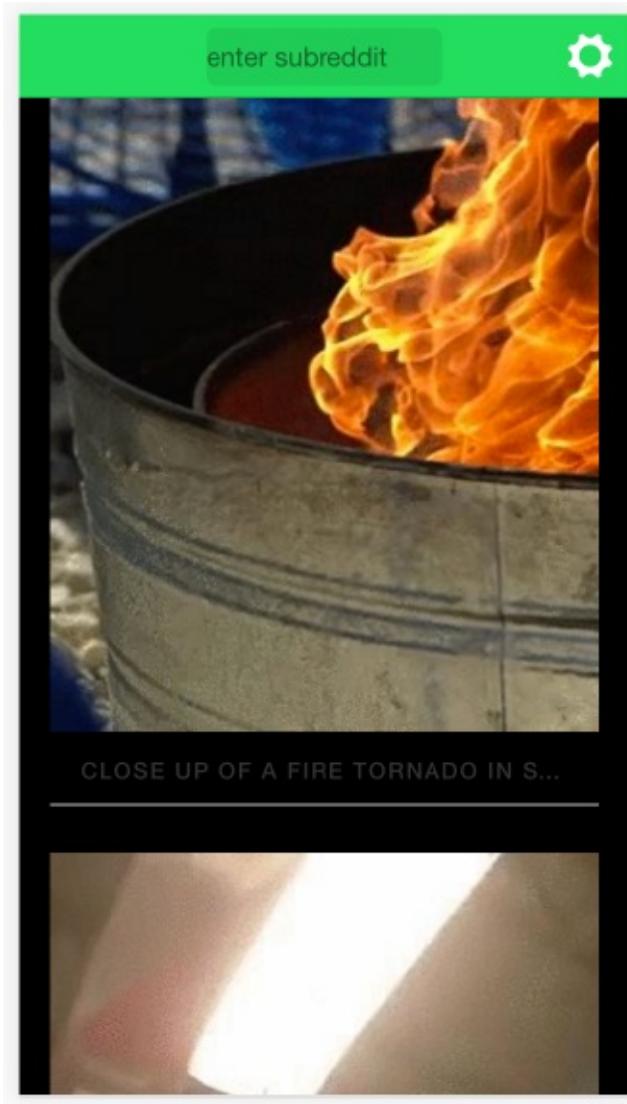
We've added a click event that will call the `playVideo` function when the video is tapped, which will pass along the `$event` (which includes data about the triggered event) and a reference to the post that was clicked. We have a separate click event for the `<ion-list-header>` that shows the GIFs title which will launch the link to the post in a Browser window.

Let's also add a bit of code to the `load` function so that the `fetchData` function gets called when we run the application (since `load` is already being called from the `ngOnInit` hook in the home page). This is just so we can see some stuff happening now, we will change it later.

> Modify the `load` function in `src/app/services/reddit.service.ts` to reflect the following:

```
load(): void {  
  
    this.fetchData();  
  
}
```

If you load up your application in the browser now, you should see something like this:



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` function.

> Modify the playVideo function in src/app/home/home.page.ts

```
playVideo(e, post): void {  
  
  console.log(e);  
  
  //Create a reference to the video  
  let video = e.target;  
  
  //Toggle the video playing  
  if(video.paused){  
  
    //Show the loader gif  
    video.play();  
  
    video.addEventListener("playing", (e) => {  
      console.log("playing video");  
    });  
  
  } else {  
    video.pause();  
  }  
  
}
```

If the user taps on the video then we want to play the video (or pause it if it is already

playing). We use the event that we passed in from the template to grab a reference to the video itself, which allows us to do things like play it and pause it.

Launching Comments in the Browser

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 title of the video we want to launch a browser with the link to the original submission of the post.

> **Modify the showComments function in `src/app/home/home.page.ts`**

```
showComments(post): void {
  Browser.open({
    toolbarColor: '#fff',
    url: 'http://reddit.com' + post.data.permalink,
    windowName: '_system'
  });
}
```

Compared to the others, this is a pretty simple function. We simply grab the link from the posts data and use Browser to launch it by calling the open method that it provides.

Loading More GIFS

One important function we haven't go to yet is the `loadMore` function, this is what is

called when the user hits the 'Load More' button. What we want to do when this happens is load in the next page of GIFs.

> **Modify the `loadMore` function in `src/app/home/home.page.ts` to reflect the following:**

```
loadMore(): void {  
  this.redditService.nextPage();  
}
```

This simply handballs the problem to our Reddit service. We will finish implementing the `nextPage` function in the Reddit service in the next lesson.

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. There are still a few loose ends that we need to tie up before everything is working properly, though.

Lesson 5: Integrating the Data Service

In this lesson, we are going to finish up the services that our application provides. First, we are going to implement our data service so that we can save and load data in the application (which we will need to use to remember the settings that a user has specified).

Then, we are going to finish implementing the remaining functions in our Reddit service. We've already done most of the heavy lifting so there isn't actually much to do here.

The Data Service

> **Modify `src/app/services/data.service.ts`** to reflect the following:

```
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('settings');
}

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

}
```

This is just a really basic implementation of a data service. All we need to do is save some data using the Ionic's Storage API on the settings key, and we also need to be able to retrieve that data.

Finishing the Reddit Service

With the data service implemented (as simple as it is) we are now going to make use of that in our Reddit service, and we are also going to finish up the remaining few functions that we are yet to implement in the service.

> **Modify the load method in `src/app/services/reddit.service.ts` to reflect the following:**

```
load(): void {
```

```
this.dataService.getData().then((settings) => {

    if(settings != null){
        this.settings = settings;
    }

    this.fetchData();

});

}
```

Previously, this method just called `fetchData()`. Now, we load in the settings from storage first. If there are any settings stored in storage, we first update our settings and then call `fetchData`. By making sure that we load in the settings first, we ensure that the `fetchData` function will be executing the appropriate request to the Reddit API.

Now we just need to deal with those few remaining methods.

> **Modify the `nextPage` method in `src/app/services/reddit.service.ts` to reflect the following:**

```
nextPage(): void {
    this.page++;
    this.fetchData();
```

```
}
```

This function will be called whenever we want to load in the next page of data. All we need to do is increment the page number, and then call `fetchData` again.

> **Modify the `resetPosts` method in `src/app/services/reddit.service.ts` to reflect the following:**

```
resetPosts(){
    this.page = 1;
    this.posts = [];
    this.after = null;
    this.fetchData();
}
```

We call this method when we want to reset everything (e.g. when we want to switch subreddits). All this does is wipe out all the data and reset the page counter.

> **Modify the `changeSubreddit` method in `src/app/services/reddit.service.ts` to reflect the following:**

```
changeSubreddit(subreddit): void {
    this.settings.subreddit = subreddit;
```

```
this.resetPosts();  
}
```

Finally, we have our `changeSubreddit` method that simply changes the `subreddit` value and then calls the `resetPosts` method.

With these finishing touches on our services, our application should pretty much be working completely now. It's not finished because we still need to implement a way to control the settings and to make it look nicer, but the functionality that is there should all work now (give it a try!).

Lesson 6: Settings

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 displaying nicely using <video> elements.

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 settings page first.

> **Modify src/app/settings/settings.page.html to reflect the following:**

```
<ion-header>
  <ion-toolbar color="secondary">
    <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 padding>

  <ion-card>
    <ion-card-header>
      About
    </ion-card-header>
    <ion-card-content>
      <p><strong>Giflist</strong> is a client for
<strong>reddit</strong> that will endlessly <strong>stream
```

GIFs from any subreddit that predominantly contains gifs. By default it uses the popular gifs subreddit, but you can change this to whatever you like, e.g: perfectloops, me_irl, chemicalreactiongifs.</p>

<p>To play a GIF, just tap it. To view the original submission for any GIF, just 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>Subreddit</h3>

```
<ion-input type="text" placeholder="enter subreddit name..."  
[(ngModel)]="redditService.settings.subreddit"></ion-input>
```

<h3>Sort</h3>

```
<ion-segment color="secondary"  
[(ngModel)]="redditService.settings.sort">  
<ion-segment-button value="/hot">  
    Hot  
</ion-segment-button>  
<ion-segment-button value="/top">  
    Top  
</ion-segment-button>
```

```

<ion-segment-button value="/new">
  New
</ion-segment-button>
</ion-segment>

<h3>Per Page</h3>

<ion-segment color="secondary"
[(ngModel)]="redditService.settings.perPage">
  <ion-segment-button value="5">
    5
  </ion-segment-button>
  <ion-segment-button value="10">
    10
  </ion-segment-button>
  <ion-segment-button value="15">
    15
  </ion-segment-button>
</ion-segment>

<ion-button expand="full" (click)="save()"
color="secondary">Save Settings</ion-button>

</ion-content>

```

There's nothing too crazy 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 } from '@angular/core';
import { ModalController } from '@ionic/angular';
import { RedditService } from '../services/reddit.service';
import { DataService } from '../services/data.service';

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

export class SettingsPage {

  constructor(public redditService: RedditService, private
  dataService: DataService, private modalCtrl: ModalController){}

}
```

```

save(): void {
    this.dataService.save({
        perPage: this.redditService.settings.perPage,
        sort: this.redditService.settings.sort,
        subreddit: this.redditService.settings.subreddit
    });

    this.close();
}

close(): void {
    this.modalCtrl.dismiss();
}

}

```

Since we are directly referring to the Reddit service in our template for the values in our inputs, they will already be set to whatever the current values for the settings are. If the user changes those settings they will automatically be updated in our Reddit service. However, we also want those settings to be saved to storage, so if the user hits the save button then we call the save method of our data service and supply the data that we want to save.

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, 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 to open the settings page, so all we need to do is finish defining the openSettings function.

> **Modify the openSettings function in `src/app/home/home.page.ts`:**

```
openSettings(): void {  
  
    this.modalCtrl.create({  
        component: SettingsPage  
    }).then((modal) => {  
  
        modal.onDidDismiss().then(() => {  
            this.redditService.resetPosts();  
        });  
  
        modal.present();  
    });  
}
```

```
});
```

```
}
```

As you can see above, we pass the Settings page 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()`.

Before we do that, we also set up an `onDidDismiss()` handler. This handler will trigger when the modal is dismissed. Any changes the user made to the settings on the settings page will be updated immediately in the Reddit service, but updating the settings won't automatically trigger a reload of the posts. So, we use this to call the `resetPosts` method so that the posts can be refetched from the Reddit API using the appropriate 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 {  
  --ion-background-color: #fff;  
}  
  
ion-item {  
  --inner-padding-end: 0;  
  --padding-start: 0;  
}  
  
ion-content button {  
  margin: 0px !important;  
}  
  
ion-spinner {  
  margin: auto;  
}  
  
ion-list-header {  
  background-color: #fff !important;  
}
```

```
margin-bottom: 0px !important;  
}  
  
video {  
  max-width: 100%;  
  height: auto;  
  margin: auto;  
  background-color: #000;  
}
```

These are all pretty basic styles, for the most part we are just changing some colours and removing padding and margins. The most important style here is the `video` style which serves to make sure the video scales and sits nicely in the list. Ideally we want the video to take up the full width of the item but in the case of portrait GIFs this isn't always possible, so in that case we would center the GIF and have a black background which looks pretty good.

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-content {  
  
  --ion-background-color: #fff !important;  
  
  ion-button button {
```

```
    margin-top: 20px !important;  
}  
  
}
```

Now we are going to define just one style that effects iOS only. Our searchbar in the header is a bit small since we've added it to the <ion-title> section, so we are going to make it a little bigger. The searchbar already displays fine on Android though, so we don't want the style to apply there.

> Add the following styles to `src/global.scss`:

```
.ios ion-title {  
  padding-left: 0;  
  padding-right: 35px;  
  padding-bottom: 5px;  
}
```

When running on iOS Ionic will automatically add a `ios` class to the <body> so we can specifically target iOS using the above method. You can do the same for Android using `md`.

Finally, we are going to add some CSS4 variables to the variables file.

> Add the following styles to the top of `src/theme/variables.scss`:

```
ion-content {  
  --ion-background-color: #1f1e1e;  
}
```

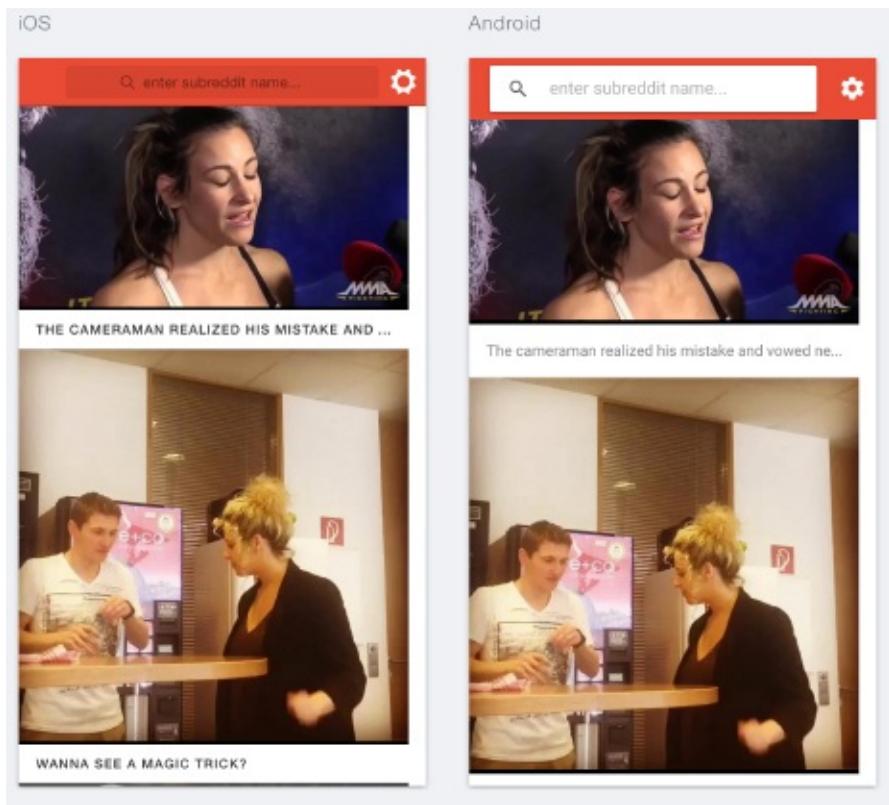
> **Modify the primary and secondary colours in `src/theme/variables.scss` (inside of `:root`):**

```
--ion-color-primary: #ecf0f1;  
  
--ion-color-secondary: #e74c3c;
```

If you load up the application should now look something like this:



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:



(ignore the white bar on the right, this isn't present on actual devices)

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 tutorial. We've learned a lot through developing this application, but the main take aways are:

- How to fetch data from a 3rd party API
- 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's a few ideas to try out:

- Create your own custom theme for the application **[EASY]**
- Create a 'Random' button that will take you to a random subreddit from a predefined array **[EASY]**
- Modify the application to return pictures instead of GIFs **[MEDIUM]**
- Modify the application to return pictures 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 everybody's favourite native plugin, 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 gotchas.

In fact, Snapaday is pretty much 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?). 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 provide web implementations of the same native functionality that you are integrating.

About Snapaday

Snapaday was actually the Ionic 1 example I used in one of my first books: **Mobile Development for Web Developers**. So, I figured when transitioning from Ionic 1 to Ionic 2 it would be a great example to use, and I've stuck with this example all the way through to Ionic 4.

The general idea 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
- 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

and here's a look at what we'll be building:



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
```

or

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

before you continue. If you run into any trouble installing Ionic or generating new projects, make sure that you have the most recent [NodeJS LTS](#) version installed. After you have that installed, you should also run the following command:

```
npm uninstall -g ionic
```

before attempting to install again.

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 snapaday blank --type=angular
```

When asked, **do not** integrate the Ionic Appflow SDK.

> Make the new project your current working directory by running the following command:

```
cd snapaday
```

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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

Create the Required Components

The structure of this application is pretty simple, so 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 few services. We will create a data service to handle storing and retrieving saved data, we will create a photo service to handle all of the operations related to taking photos, and since we'll be using quite a lot of alerts in the application we will be creating a service to handle those more easily.

> Run the following command to generate a Data service:

```
ionic g service services/Data
```

NOTE: Notice that we use services/Data instead of just Data 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 a Photo service:

```
ionic g service services/Photo
```

> 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. 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
```

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 { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', loadChildren:
```

```
'./home/home.module#HomePageModule' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  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 settings 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

Capacitor is not included by default in our applications, so we need to set it up. You should still complete this step even if you don't intend to build for iOS and Android because we will also be using Capacitor for the web platform.

To set up Capacitor in an Ionic application, all you need to do is run a single command.

> Run the following command to enable Capacitor:

```
ionic integrations enable capacitor
```

This will handle setting up the Capacitor CLI and Capacitor Library in your project.

Keep in mind that this will give your application a default 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 now. So, you may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`.

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. 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
```

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your application code and dependencies over 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.

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'
})
export class AppComponent {

  constructor() {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });
  }
}
```

All we are doing here is hiding the StatusBar but you can use this API to style the Status Bar however you like. 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 likely to get wiped by the operating system).

To enable Ionic storage, you need to install it.

> **Install Ionic Storage with the following command:**

```
npm install @ionic/storage --save
```

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 { RouterModule, RouteReuseStrategy, Routes } 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 and entryComponents. 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 { RouterModule } from '@angular/router';

import { HomePage } from './home.page';
import { SlideshowPage } from '../slideshow/slideshow.page';
import { DaysAgoPipe } from '../pipes/days-ago.pipe';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild([
      {
        path: '',
        component: HomePage
      }
    ])
  ],
  declarations: [HomePage, SlideshowPage, DaysAgoPipe],
  entryComponents: [SlideshowPage]
})
```

```
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 --save
```

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 **SocialSharing** plugin:

```
npm install cordova-plugin-x-socialsharing --save
```

```
npm install @ionic-native/social-sharing@beta --save
```

We are installing the **SocialSharing** plugin so that the user will be able to share their photo after they take it. We will be using Ionic Native to interact with this Cordova plugin (remember, you can use Cordova plugins and Ionic Native with Capacitor if you want to). Since we are using Ionic Native, we have to install the Ionic Native package for the plugin. At the moment, it is important to include the `@beta` tag as we need to use the latest

version of the Ionic Native package. However, eventually you will be able to get rid of the @beta tag (if the install command is no longer working, you should remove it).

> Run the following commands to add the LocalNotifications plugin:

```
npm install cordova-plugin-local-notification --save
```

```
npm install @ionic-native/local-notifications@beta --save
```

We are installing the **LocalNotifications** plugin so that we can send notifications to the user each day to remind them to take a photo.

> Run the following command to add the Ionic Native package for the WebView:

```
npm install @ionic-native/ionic-webview@beta --save
```

The Ionic web view provides a helper function that can convert local file URLs into a useable format for us, we will need to make use of this later. Note that we don't actually need to install the web view plugin itself because the web view is already installed by default.

Since we are using plugins from Ionic Native, we will need to set them up as providers in our 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 { RouterModule, RouteReuseStrategy, Routes } 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 { SocialSharing } from '@ionic-native/social-sharing/ngx';
import { LocalNotifications } from '@ionic-native/local-
notifications/ngx';
import { WebView } from '@ionic-native/ionic-webview/ngx';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot(),
IonicStorageModule.forRoot(), AppRoutingModule],
  providers: [
    SocialSharing,
    LocalNotifications,
    WebView,
    { provide: RouteReuseStrategy, useClass: IonicRouteStrategy }
  ],
  bootstrap: [AppComponent]
})
```

```
export class AppModule {}
```

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 even 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 crazy 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:



The trickiest part of this layout is that "SMILE!" 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.html to reflect the following:**

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>
      
    </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>
        
        <ion-badge slot="end" color="light">0 days ago</ion-badge>
      </ion-item>
      <ion-item-options>
        <ion-item-option tappable color="light" (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>
      
    </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, of course).

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>
        
        <ion-badge slot="end" color="light">0 days ago</ion-
      badge>
    </ion-item>
  </ion-list>
</ion-content>
```

```
<ion-item-options>
  <ion-item-option tappable 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 turn this into a loop for all the photos), 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 of course change that later.

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 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 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="!photoTaken" tappable (click)="takePhoto()">
      
    </ion-item>

    <ion-item-sliding>

      <ion-item>
        
        <ion-badge slot="end" color="light">0 days ago</ion-
badge>
      </ion-item>

      <ion-item-options>
```

```

<ion-item-option tappable 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 `tappable` attribute . Visually, this changes nothing, but on mobile everything that is not a `<button>` or `<a>` that has a click handler will have a slight tap delay. We don't want to have this delay so we add the `tappable` attribute.

We also have a click handler set up to trigger the `takePhoto` function 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="!photoTaken"
```

the extra button we added will only display if the photoTaken value is `false`, since we are essentially saying "display this button if the photoTaken variable is not true". Later on, we will create a **photoTaken** variable in **home.page.ts** which will keep track of whether a user has already taken a photo on the current day or not.

If you take a look at the application now by running:

```
ionic serve
```

you should see something like this:



Obviously, there is a lot left to do but that's all we need to do for now for the home page, so let's move on to the slideshow page.

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)="playPhotos()"><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!

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!

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

So this is going to be a pretty large lesson. We will have Ionic Native available to us in this application, which if you remember from the basics section is a service created by Ionic that wraps Cordova plugins to make using them a bit easier (e.g. plugins like the Camera). However, Capacitor also comes with its own set of Core APIs that we can use, which includes an API for both the Camera and the Filesystem.

So, what do we use? It does not particularly matter which you use if you would prefer to just do everything with Ionic Native you can do that, but if you want to use the Capacitor APIs you can do that too. The Ionic team has put some effort into designing easier to use APIs in Capacitor, and so I generally prefer to use the Capacitor APIs where available - we will be doing that for the Camera and Filesystem functionality in this lesson.

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 a lot of different alerts in this application because there are many different places where errors can occur that the users needs to be notified about (taking the photo, moving the photo) and we will also be alerting the user when things have gone well as well.

To create an alert, the syntax looks this:

```
this.alertCtrl.create({
  header: 'My header',
  message: 'My message',
  buttons: [
    {
      text: 'Ok'
    }
  ]
}).then((alert) => {
  alert.present();
});
```

Which is quite a bit of code, and if we are going to be calling this multiple times in the same file it's going to get pretty messy. So we're going to create a service which will handle creating the alert for us. Once we have created it, we will instead be able to create an alert by doing this:

```
this.simpleAlert.createAlert('Oops!', 'Some  
message').then((alert) => {  
  alert.present();  
});
```

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.

> **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){  
  
  }  
  
  createAlert(title: string, message: string): Promise<any> {  
  
    return this.alertCtrl.create({
```

```
        header: title,  
        message: message,  
        buttons: [  
          {  
            text: 'Ok'  
          }  
        ]  
      );  
  
    }  
  
  }  

```

We've imported the `AlertController` service here so that we can create an alert, and then we have just created a single function that takes in a title and a message and returns an alert using those values. This just creates the alert for us, we will still need to display it ourselves by presenting it wherever we need it.

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 simpleAlert: 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.simpleAlert`.

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 add a few variables to our constructor and, so we don't need to constantly modify our imports, import all of the things we will require for the rest of the application.

> **Modify src/app/home/home.ts to reflect the following:**

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';
import { ModalController, AlertController, LoadingController,
IonList } from '@ionic/angular';
import { PhotoService } from '../services/photo.service';
import { SimpleAlertService } from '../services/simple-
alert.service';
import { SlideshowPage } from '../slideshow/slideshow.page';

import { SocialSharing } from '@ionic-native/social-sharing/ngx';

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

```
@ViewChild(IonList) slidingList: IonList;

constructor(
  public photoService: PhotoService,
  private alertCtrl: AlertController,
  private simpleAlert: SimpleAlertService,
  private modalCtrl: ModalController,
  private socialSharing: SocialSharing,
  private loadingCtrl: LoadingController,
  public sanitizer: DomSanitizer
){}

ngOnInit(){

}

takePhoto(): void {

}

playSlideshow(): void {

}
```

```
    deletePhoto(photo): 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, including the `SocialSharing` plugin from Ionic Native (since there isn't a Capacitor API for that, we will be using the Social Sharing Cordova plugin through Ionic Native). Eventually, we will be making calls from this home page to the photos service that we will create in just a moment. We will trigger the loading of photos through `ngOnInit` (which runs as soon as this component is initialised), we will trigger the actual taking of the photo through `takePhoto`, 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. Let me explain.

Basically, 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 WebView 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 { DataService } from './data.service';
import { Platform } from '@ionic/angular';
import { Plugins, CameraResultType, CameraSource,
FilesystemDirectory } from '@capacitor/core';
import { WebView } from '@ionic-native/ionic-webview/ngx';

interface Photo {
  name: string,
```

```
    path: string,  
    dateTaken: Date  
}  
  
const { Camera, Filesystem } = Plugins;  
  
@Injectable({  
  providedIn: 'root'  
})  
export class PhotoService {  
  
  public photos: Photo[] = [];  
  public loaded: boolean = false;  
  
  public photoTaken: boolean = false;  
  
  constructor(private dataService: DataService, private  
platform: Platform, private webView: WebView){  
  
}  
  
  load(): void {  
  
    // Uncomment to use test data  
    /*this.photos = [  
  
      { name: 'test', path: 'https://placehold.it/100x100',  
dateTaken: new Date(2018,5,5) },
```

```

        { name: 'test', path: 'https://placehold.it/100x100',
dateTaken: new Date(2018,5,6) },
        { name: 'test', path: 'https://placehold.it/100x100',
dateTaken: new Date(2018,5,8) },
        { name: 'test', path: 'https://placehold.it/100x100',
dateTaken: new Date(2018,5,10) }

    ]*/
}

this.platform.resume.subscribe(() => {

    if(this.photos.length > 0){

        let today = new Date();
        let lastDate = new
Date(this.photos[0].dateTaken);

        if(lastDate.setHours(0,0,0,0) ===
today.setHours(0,0,0,0)){
            this.photoTaken = true;
        } else {
            this.photoTaken = false;
        }
    }

});
}

```

```
takePhoto(): Promise<any> {
    return Promise.resolve(true);
}

createPhoto(name, path): void {

}

deletePhoto(photo): void {

}

save(): void {

}

}
```

We've also just set up a skeleton for our photos service for now, but there is a little bit more going on here already. We've got some pretty standard imports, including our data service which we will implement a little later. We also set up a simple interface:

```
interface Photo {
    name: string,
```

```
    path: string,  
    dateTaken: Date  
}
```

We will be using this as the type for the photo objects we are going to create. 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.

Unlike with Ionic Native, for Capacitor plugins we first import Plugins from @capacitor/core and then we set up a reference to the plugins we are interested in like this:

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

We can then directly access Camera and Filesystem throughout the class. We've added a member variable loaded to keep track of whether the photos have been loaded from storage yet (which is something we will do in the next lesson), photoTaken to tell if a photo has already been taken today and photos to hold all of our photo data.

So that we have some data to work with, 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 remove it later though!).

We've also added a weird resume listener here. 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 photoTaken 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 logic we will run later in our loadPhotos function won't run to determine if the photo was taken or not, because the application is just being resumed not reloaded. 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 photoTaken variable according to that.

Now we're going to implement the takePhoto function which will handle taking the photo, so let's create a basic version of that now.

> **Modify the takePhoto function in photo.service.ts to reflect the following:**

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

    return new Promise((resolve, reject) => {

        if(!this.loaded || this.photoTaken){
            reject('Not allowed to take photo');
        }

        let options = {
```

```

        quality: 50,
        width: 600,
        allowEditing: false,
        resultType: CameraResultType.Uri,
        source: CameraSource.Camera
    };

    Camera.getPhoto(options).then(
        (photo) => {
            console.log(photo)
        }, (err) => {
            console.log(err);
            reject('Could not take photo');
        }
    );
}

}

```

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, then the rest of the function will not finish executing.

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. In this case, we want to use the camera 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).

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 trigger an error by rejecting the promise.

The image that is taken with the camera, and the path that is returned, is in a temporary storage folder. So 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 this if it is critical that the photo is never lost). 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` function to do the following:

- Use the `imagePath` 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 function in photo.service.ts to reflect the following:**

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

    return new Promise((resolve, reject) => {

        if(!this.loaded || this.photoTaken){
            reject('Not allowed to take photo');
        }

        let options = {
            quality: 50,
            width: 600,
            allowEditing: false,
            resultType: CameraResultType.Uri,
            source: CameraSource.Camera
        };

        Camera.getPhoto(options).then(
            (photo) => {

                Filesystem.readFile({
```

```
        path: photo.path
    }).then((result) => {

        let date = new Date(),
            time = date.getTime(),
            fileName = time + '.jpeg';

        Filesystem.writeFile({
            data: result.data,
            path: fileName,
            directory: FilesystemDirectory.Data
        }).then((result) => {

            Filesystem.getUri({
                directory: FilesystemDirectory.Data,
                path: fileName
            }).then((result) => {
                console.log(result);
                let filePath =
this.webView.convertFileSrc(result.uri);

                this.createPhoto(fileName, filePath);
                resolve(result.uri);
            }, (err) => {
                console.log(err);
                reject('Could not find photo in
storage');
            });
        });
    });
});
```

```

        }, (err) => {
            console.log(err);
            reject('Could not write photo to storage');
        });

    }, (err) => {
        console.log(err);
        reject('Could not read photo data');
    });

}, (err) => {
    console.log(err);
    reject('Could not take photo');
}

);

});

}

```

That's quite a bit more complicated now. The code above 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. Use the `getUri` method to return a URI of the file we just created.
4. Send that URI to the `createPhoto` method after converting the path into a Capacitor friendly format

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 know where to find it later.

> Modify the `createPhoto` function to reflect the following:

```
createPhoto(name, path): void {  
  
    this.photos.unshift({  
        name: name,  
        path: path,  
        dateTaken: new Date()  
    });  
  
    this.save();  
  
}
```

As you can see it's pretty simple. We just create a new photo object, and then we add it to our `this.photos` array. Rather than using the `push` method to add it to the array (which adds it to the end) we add it to the start of the array with `unshift`. We do this because we want to display the photos in reverse order (most recent first) and doing it this way makes our lives easier later.

We also have a call to the `save` function, which will save our data into storage but we haven't implemented that functionality yet. 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` function to reflect the following:**

```
deletePhoto(photo): void {  
  
    // Remove data from storage  
    let index = this.photos.indexOf(photo);  
  
    if(index > -1){  
        this.photos.splice(index, 1);  
        this.save();  
    }  
  
    // If the delete photo was taken today, allow the user to  
    // take a new photo
```

```

let today = new Date();

if(photo.dateTaken.setHours(0,0,0,0) ===
today.setHours(0,0,0,0)){
    this.photoTaken = false;
}

// Remove from filesystem
Filesystem.deleteFile({
    path: photo.name,
    directory: FilesystemDirectory.Data
}).then((result) => {

    console.log(result);

}, (err) => {
    console.log(err);
});

}

```

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, and the directory that it was stored in.

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. Since you're likely testing through the browser, 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.html` to reflect the following:**

```
<ion-header>
  <ion-toolbar color="danger">
    <ion-title>
      
    </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.photoTaken" tappable
      (click)="takePhoto()">
```

```


</ion-item>

<ion-item-sliding *ngFor="let photo of photoService.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 tappable 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 photoService.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 `` 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:**

```
deletePhoto(photo): void {
  this.slidingList.closeSlidingItems().then(() => {
    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:**

```
takePhoto(): void {  
  
    this.loadingCtrl.create({  
        message: 'Saving Photo...',  
    }).then((overlay) => {  
  
        overlay.present();  
  
        this.photoService.takePhoto().then((photo) => {  
  
            overlay.dismiss();  
  
            this.alertCtrl.create({  
                header: 'Nice one!',  
                message: 'You\'ve taken your photo for today, would you  
also like to share it?',  
                buttons: [  
                    {  
                        text: 'No, Thanks'  
                ]  
            }).present();  
        });  
    });  
}
```

```

        },
        {
            text: 'Share',
            handler: () => {
                console.log(photo);
                this.socialSharing.share('I\'m taking a selfie
every day with #Snapaday', null, photo, null)
            }
        }
    ]
}).then((prompt) => {
    prompt.present();
});

}, (err) => {
    overlay.dismiss();
    this.simpleAlert.createAlert('Oops!', err).then((alert)
=> {
    alert.present();
});
});

});
}

```

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 `overlay.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. So, we pop up a custom alert to give them that option, and if they choose the Share option we trigger the `SocialSharing` plugin that we are importing from Ionic Native. 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 (Facebook, GMail, Messenger, and so on).

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 give it a try for yourself you'll need to run the application on a device with a camera. If you don't know how to do that 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 to be able to test the application now, you will need to **temporarily** add:

```
this.loaded = true;
```

to the load method in the photo service, and you will need to call the load method from your home page:

```
ngOnInit(){
  this.photoService.load();
}
```

You can't take a photo until the data has finished loading, so we can just fake that for now if you are eager to test out the camera. In the next lesson, we'll be looking at how to create a data service to store our photos data permanently, so that it is available when the user comes back to the application the next time.

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 learn how to create a data storage service to both save data to permanent storage, and to load it into the application.

We've already got a lot of this process set up, we've already generated a 'Data' service which we have imported into our photo service, and we even have a save function created that we are calling whenever we want data saved. So all we really have to do is:

- Implement the save function so that it calls the data service
- Modify the empty Data service to provide save and load functionality
- Load the photo data into the application when the application is opened

Implementing the Data Service

We're going to add the code for `data.service.ts` now so that it will save into storage any data that it is passed. The code for this service is actually surprisingly simple, so let's take a look at it first and then talk through it.

> **Modify `src/app/services/data.service.ts` to reflect the following:**

```
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('photos');
  }

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

Let's take a look at the import at the top of this file:

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

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.

Let's take a look at the `getData` function:

```
getData(): Promise<any> {
  return this.storage.get('photos');
}
```

This function will allow us to retrieve the latest data that has been stored. The `get` method of our storage will return a promise, but notice that we are not setting up the handler for when the promise finishes here, instead we just return the result of the `get` method directly. This allows us to set up the handler from wherever in the code this method is being called, which makes more sense for the flow of the application (hopefully this will be

made more clear shortly).

Then we have our save function, which handles actually saving the data into storage:

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

This will store our photos array under the key 'photos' in storage. That's all there is to saving the data, which isn't really all that complex. Now we just need to handle loading that data back into the application. We're going to do this by modifying the load method in our photo service.

> Modify the load function in `src/app/services/photo.service.ts` to reflect the following:

```
load(): void {  
  
  // Uncomment to use test data  
  /*this.photos = [  
  
    { name: 'test', path: 'https://placehold.it/100x100',  
    dateTaken: new Date(2018,5,5) },  
    { name: 'test', path: 'https://placehold.it/100x100',  
    dateTaken: new Date(2018,5,5) }  
  ]*/  
}
```

```

dateTaken: new Date(2018,5,6) },
    { name: 'test', path: 'https://placehold.it/100x100' ,
dateTaken: new Date(2018,5,8) },
    { name: 'test', path: 'https://placehold.it/100x100' ,
dateTaken: new Date(2018,5,10) }

]*/

```

```

this.platform.resume.subscribe() => {

    if(this.photos.length > 0){

        let today = new Date();
        let lastDate = new
Date(this.photos[0].dateTaken);

        if(lastDate.setHours(0,0,0,0) ===
today.setHours(0,0,0,0)){
            this.photoTaken = true;
        } else {
            this.photoTaken = false;
        }
    }

});

this.dataService.getData().then((photos) => {

```

```

        console.log(photos);

        if(photos != null){
            this.photos = photos;
        }

        if(this.photos.length > 0){

            let today = new Date();
            let lastDate = new Date(this.photos[0].dateTaken);

            if(lastDate.setHours(0,0,0,0) ===
today.setHours(0,0,0,0)){
                this.photoTaken = true;
                this.photoTaken = false; //TESTING ONLY - This
will allow you to take more than one photo per day
            }

        }

        this.loaded = true;

    });

}


```

In the section we have added at the bottom, we are calling the `getData()` function we

just implemented in our data service. This will return all of the photo data that we have in storage. However, there may not be anything in storage yet, so we first check if the returned photos are null before setting them on the `this.photos` class member.

After loading the photos, we do a little trickery to determine whether or not the user is allowed to take a photo today. We simply check the date of the photo in the first position and compare it to the current date, if they match then we set the `photoTaken` variable to true. For now, we have an extra testing line set so that it will always allow us to take a photo, this should be removed if you want to enable the "one photo per day" restriction>.

Once all this has finished executing, we set the `this.loaded` flag to true to indicate that the photos are loaded and ready to go. In order to trigger this load method, we will need to make a call to it from somewhere. We will do this in the `ngOnInit` method in our home page.

> **Modify the `ngOnInit` hook in `src/app/pages/home/home.ts` to reflect the following:**

```
ngOnInit(){
  this.photoService.load();
}
```

Now all we have left to do is make our save function actually make a call to the data service.

> Modify the save function in `src/app/services/photo.service.ts` to reflect the following:

```
save(): void {  
    this.dataService.save(this.photos);  
}
```

Now whenever the save function 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.

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, args?) {
    let now = new Date();
    let takenDate = new Date(value);
    let oneDay = 24 * 60 * 60 * 1000;
    let diffDays = Math.round(Math.abs((takenDate.getTime() - now.getTime())/(oneDay)));
    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, which is why we use args? because that parameter is not required.

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` function so that we can actually open the page.

> **Modify the `playSlideshow` function in `src/app/home/home.page.ts` to reflect the following:**

```
playSlideshow(): void {  
  
  if(this.photoService.photos.length > 1){  
  
    this.modalCtrl.create({  
      component: SlideshowPage  
    }).then((modal) => {  
      modal.present();  
    });  
  
  } else {  
    // ...  
  }  
}
```

```
        this.simpleAlert.createAlert('Oops!', 'You need at least  
two photos before you can play a slideshow.').then((alert) => {  
    alert.present();  
});  
  
}  
  
}
```

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, ElementRef, ViewChild, OnInit } from
'@angular/core';
import { SafeResourceUrl, DomSanitizer } from '@angular/platform-
browser';
import { ModalController } from '@ionic/angular';
import { PhotoService } from '../services/photo.service';

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

  private imagePlayerInterval: any;
  public imageSrc: SafeResourceUrl;

  constructor(private photoService: PhotoService, private
modalCtrl: ModalController, private sanitizer: DomSanitizer){

}

ngOnInit(){
  this.playPhotos();
}

playPhotos(): void {

```

```

let i = 0;

//Clear any interval already set
clearInterval(this.imagePlayerInterval);

//Restart
this.imagePlayerInterval = setInterval(() => {
  if(i < this.photoService.photos.length){
    this.imageSrc =
      this.sanitizer.bypassSecurityTrustUrl(this.photoService.photos[i].p

      i++;
  } else {
    clearInterval(this.imagePlayerInterval);
  }
}, 500);

}

close(): void {
  this.modalCtrl.dismiss();
}

}

```

The most important function here is the playPhotos function. 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 isn't photos left to display then the interval is cleared. In the code above the interval is set to run every 500ms or every 0.5 seconds, 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.

The `playPhotos` function is triggered automatically by the use of the `ngOnInit` hook. The only other function 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.

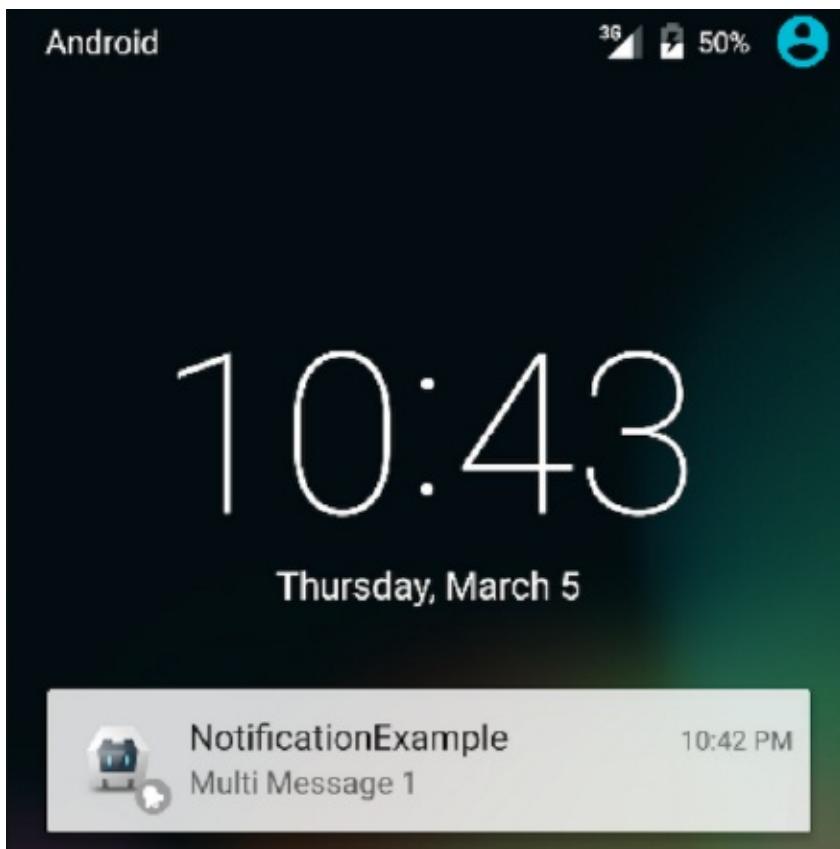
Lesson 7: Integrating Local Notifications

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 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 { Platform } from '@ionic/angular';
import { LocalNotifications, ELocalNotificationTriggerUnit } from
'@ionic-native/local-notifications/ngx';

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {

  constructor(private platform: Platform, private
localNotifications: LocalNotifications) {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });

    if(this.platform.is('cordova')){
```

```
this.localNotifications.isScheduled(1).then( (scheduled) =>
{
  if(!scheduled){

    let firstNotificationTime = new Date();

firstNotificationTime.setHours(firstNotificationTime.getHours() + 24)

    this.localNotifications.schedule({
      id: 1,
      title: 'Snapaday',
      text: 'Have you taken your snap today?',
      trigger: {
        at: firstNotificationTime,
        every: ELocalNotificationTriggerUnit.DAY
      }
    });

  }
}

});
```

```
}
```

We can access the functionality for local notifications through `LocalNotifications` from Ionic Native. We first check if a notification with the id of 1 is already scheduled, if it is we do nothing, but if it's not then we create a new notification using the `schedule` method.

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 (like we just did when checking to see if it was already scheduled). 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 every week etc.).

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's 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 theming in the basics sections.

> Add the following rule to the `theme/variables.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 {  
  margin: 0 !important;  
}  
  
ion-title img {  
  max-height: 39px;  
}  
  
ion-item {  
  
  margin-bottom: 2px;  
  --inner-padding-end: 0px;  
  --padding-start: 0px;  
  
  img {  
    width: 100%;  
    height: auto;  
  }  
}  
  
ion-badge {  
  position: absolute;  
  right: 0px;  
  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 flexbox to get that to happen. Flexbox is a little more advanced, but 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 like this:



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 tutorial. We've learned a lot through developing this application, but the main take aways 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 provider
- 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's 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 - one important example being that we will have lots of different bits of data we need to save, rather than just one set of data, so our data service is going to be more complex than the ones we have been creating in other applications.

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:

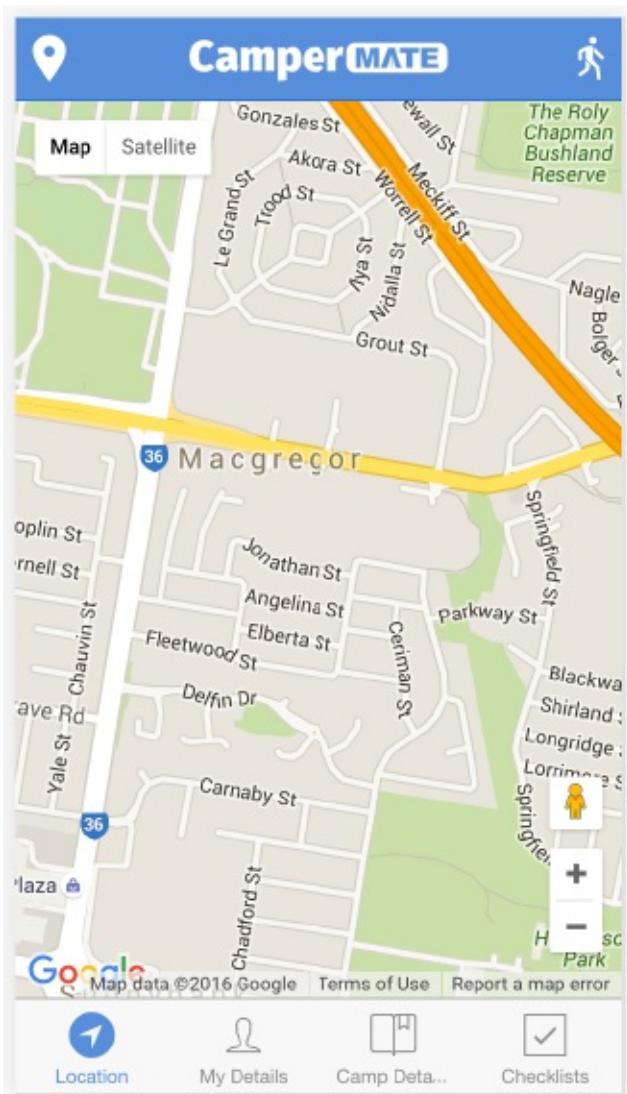
- A map that the user can set their camp location on. When they have left their campsite they will be able to click a button which will then show them how to get back to their campsite.
- A form that will allow the user to store personal details relevant to camping (car and trailer registration etc.)

- 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:



CamperMATE

My Details

Update this form with your details so you have an easy reference for later.

Car Registration
IAMCOOL

Trailer Registration
TRAILER

Trailer Dimensions
8 x 16ft

Phone Number
555444333

Notes

 Location  My Details  Camp Data...  Checklists

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
```

or

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

before you continue. If you run into any trouble installing Ionic or generating new projects, make sure that you have the most recent [NodeJS LTS](#) version installed. After you have that installed, you should also run the following command:

```
npm uninstall -g ionic
```

before attempting to install again.

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 campermate blank --type=angular
```

When asked, **do not** integrate the Ionic Appflow SDK.

> 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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

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 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 as part of 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.

> Run the following command to install types for Google Maps

```
npm install @types/google-maps --save
```

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
```

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 example. 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 { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: '', loadChildren: './home/home.module#HomePageModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})

export class AppRoutingModule {}
```

Even though we have multiple pages in our application, we are only defining the default route here and directing it to the home page module. The home page will host our tabs, and we will define additional routes in the home page module.

We are going to create a separate home-routing.module.ts file to hold the routes for the 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:
              '../location/location.module#LocationPageModule'
          }
        ]
      },
      {
        path: 'camp',
        children: [
          {
            path: '',
            loadChildren:
              '../camp/camp.module#CampPageModule'
          }
        ]
      }
    ]
  }
];
```

```

        loadChildren: '../camp-details/camp-
details.module#CampDetailsPageModule'
    }
]
},
{
    path: 'me',
    children: [
    {
        path: '',
        loadChildren: '../my-details/my-
details.module#MyDetailsPageModule'
    }
]
}
],
},
{
    path: '',
    redirectTo: '/tabs/location',
    pathMatch: 'full'
}
];
}

@NgModule({
    imports: [RouterModule.forChild(routes)],
    exports: [RouterModule]
})

```

```
export class HomePageRoutingModule { }
```

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

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

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 the

next lesson.

We will also be making use of the `ReactiveFormsModule` in both our `CampDetailsPage` and the `MyDetails` Page. 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 { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';

import { CampDetailsPage } from './camp-details.page';

const routes: Routes = [
  {
    path: '',
    component: CampDetailsPage
  }
];

@NgModule({
```

```

imports: [
  CommonModule,
  ReactiveFormsModule,
  IonicModule,
  RouterModule.forChild(routes)
],
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 { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';

import { MyDetailsPage } from './my-details.page';

const routes: Routes = [
{
  path: '',
  component: MyDetailsPage
}

```

```
];

@NgModule({
  imports: [
    CommonModule,
    ReactiveFormsModule,
    IonicModule,
    RouterModule.forChild(routes)
  ],
  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 { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';
```

```
import { LocationPage } from './location.page';
import { GoogleMapComponent } from '../components/google-
map/google-map.component';

const routes: Routes = [
  {
    path: '',
    component: LocationPage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild(routes)
  ],
  declarations: [LocationPage, GoogleMapComponent]
})
export class LocationPageModule {}
```

Configure Capacitor

Capacitor is not included by default in our applications, so we need to set it up. You should still complete this step even if you don't intend to build for iOS and Android

because we will also be using Capacitor for the web platform.

To set up Capacitor in an Ionic application, all you need to do is run a single command.

> **Run the following command to enable Capacitor:**

```
ionic integrations enable capacitor
```

This will handle setting up the Capacitor CLI and Capacitor Library in your project.

Keep in mind that this will give your application a default 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 now. So, you may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`.

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. 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
```

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your application code and dependencies over 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.

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'
})
export class AppComponent {

  constructor() {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });
  }
}
```

All we are doing here is hiding the StatusBar but you can use this API to style the Status Bar however you like. 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 likely to get wiped by the operating system).

To enable Ionic storage, you need to install it.

> **Install Ionic Storage with the following command:**

```
npm install @ionic/storage --save
```

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 { RouterModule, RouteReuseStrategy, Routes } 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 --save
```

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 even 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 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>Camp Details</ion-label>
    </ion-tab-button>
```

```
<ion-tab-button tab="camp">
  <ion-icon name="bookmarks"></ion-icon>
  <ion-label>My Details</ion-label>
</ion-tab-button>

</ion-tab-bar>

</ion-tabs>
```

The implementation of tabs have been simplified tremendously in the final version of Ionic

4. Earlier on during Ionic 4's development, we had to manually set up multiple different router outlets and configure them using rather confusing syntax (like `(outlet: path)`).

This style of tabs also did not support lazy loading of tabs. If you have already implemented tabs using the kind of syntax I am referring to, you'll be glad to know it is now much easier and this section of the book is now *much* shorter. If this is your first experience with Ionic, or you otherwise haven't been exposed to this, you can just forget this paragraph.

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 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:  
'../location/location.module#LocationPageModule'  
}  
]  
},  
{  
    path: 'camp',  
    children: [  
        {  
            path: '',  
            loadChildren: '../camp-details/camp-  
details.module#CampDetailsPageModule'  
        }  
    ]  
},  
{  
    path: 'me',  
    children: [  
        {  
            path: '',  
            loadChildren: '../my-details/my-  
details.module#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:  
        '.../location/location.module#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 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. 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>
  <ion-toolbar color="primary">
    <ion-title>
      
    </ion-title>
    <ion-buttons slot="start">
      <ion-button (click)="setLocation()"><ion-icon slot="icon-only" name="pin"></ion-icon></ion-button>
    </ion-buttons>
    <ion-buttons slot="end">
      <ion-button (click)="takeMeHome()"><ion-icon slot="icon-only" name="walk"></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.

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 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" functions.

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, ViewChild, OnInit } from '@angular/core';
import { AlertController, LoadingController, Platform } from
  '@ionic/angular';
import { Plugins } from '@capacitor/core';
```

```
import { GoogleMapComponent } from '../components/google-
map/google-map.component';
import { DataService } from '../services/data.service';

const { Geolocation } = Plugins;

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

  //@ViewChild(GoogleMapComponent) map: GoogleMapComponent;

  private latitude: number;
  private longitude: number;

  constructor(
    private alertCtrl: AlertController,
    private loadingCtrl: LoadingController,
    private dataService: DataService,
    private platform: Platform
  ){
  }

  ngOnInit(){

```

```
}

setLocation(): 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 couple of services 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. We also declare two additional member variables, `latitude` and `longitude`, which we will use to hold the user's location later. Finally, we add the two functions that the buttons in the template call, but we will implement their functionality 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) 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 functions 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) 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>
  <ion-toolbar color="primary">
    <ion-title>
      
```

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

<ion-content padding>

<ion-card>

  <ion-card-header>
    Camp Details
  </ion-card-header>

  <ion-card-content>
    Update this form with the details of your current camp so
    you have an easy reference for later.
  </ion-card-content>

</ion-card>

<ion-list lines="none">

  <ion-item>
    <ion-label position="stacked">Gate Access Code</ion-
    label>
    <ion-input type="text"></ion-input>
  </ion-item>

  <ion-item>
```

```
<ion-label position="stacked">Ammenities Code</ion-
label>
    <ion-input type="text"></ion-input>
</ion-item>

<ion-item>
    <ion-label position="stacked">WiFi Password</ion-
label>
    <ion-input type="text"></ion-input>
</ion-item>

<ion-item>
    <ion-label position="stacked">Phone Number</ion-
label>
    <ion-input type="text"></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 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. We use an `<ion-card>` to create a little header area for the page where we can describe what the page is for, but this is purely decoration. 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.

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, Validators } from
  '@angular/forms';
import { DataService } from '../services/data.service';

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

  public campDetailsForm: FormGroup;

  constructor(private formBuilder: FormBuilder, private
  dataService: DataService) {

  }
}
```

```
ngOnInit(){

}

saveForm(): void {
}

}
```

The only thing we're doing here that's out of the ordinary is importing **FormBuilder** and **Validators** 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 (and ControlGroup) 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'll be getting to that a little later though.

We've also added a `saveForm` function which we will use later to save the values the user enters in the form. 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>
```

```
<ion-toolbar color="primary">
  <ion-title>
    
  </ion-title>
</ion-toolbar>
</ion-header>

<ion-content padding>

  <ion-card>

    <ion-card-header>
      My Details
    </ion-card-header>

    <ion-card-content>
      Update this form with your details so you have an easy
      reference for later.
    </ion-card-content>

  </ion-card>

  <ion-list lines="none">

    <ion-item>
      <ion-label position="stacked">Car Registration</ion-
      label>
      <ion-input type="text"></ion-input>
    </ion-item>
  </ion-list>
</ion-content>
```

```
</ion-item>

<ion-item>
  <ion-label position="stacked">Trailer
Registration</ion-label>
  <ion-input type="text"></ion-input>
</ion-item>

<ion-item>
  <ion-label position="stacked">Trailer
Dimensions</ion-label>
  <ion-input type="text"></ion-input>
</ion-item>

<ion-item>
  <ion-label position="stacked">Phone Number</ion-
label>
  <ion-input type="text"></ion-input>
</ion-item>

<ion-item>
  <ion-label position="stacked">Notes</ion-label>
  <ion-textarea 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, Validators } from
'@angular/forms';
import { DataService } from '../services/data.service';

@Component({
  selector: 'app-my-details',
  templateUrl: './my-details.page.html',
  styleUrls: ['./my-details.page.scss']
})

export class MyDetailsPage implements OnInit {

  public myDetailsForm: FormGroup;

  constructor(private formBuilder: FormBuilder, private
  dataService: DataService) {

  }

  ngOnInit(){
    this.myDetailsForm = this.formBuilder.group({
      name: [null, Validators.required],
      address: [null, Validators.required],
      city: [null, Validators.required],
      state: [null, Validators.required],
      zip: [null, Validators.required],
      phone: [null, Validators.required],
      email: [null, Validators.required]
    });
  }

  saveDetails() {
    const details = this.myDetailsForm.value;
    this.dataService.addCamp(details);
    this.myDetailsForm.reset();
  }
}
```

```
}

saveForm(): void {

}

}
```

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 Form Builder 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 a pretty easy way to go about managing user input, but if you have more large and complex forms it starts to become easier to use **Form Builder**.

As well as making the code a bit nicer, by using Form Builder you can manage each of your input 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" (input)="saveForm()">

    <ion-item>
      <ion-label position="stacked">Gate Access Code</ion-label>
      <ion-input formControlName="gateAccessCode" type="text"></ion-input>
    </ion-item>

    <ion-item>
      <ion-label position="stacked">Ammenities Code</ion-label>
      <ion-input formControlName="ammnenitiesCode" type="text"></ion-input>
    </ion-item>

  </form>
</ion-list>
```

```

</ion-item>

<ion-item>
  <ion-label position="stacked">WiFi Password</ion-
label>
  <ion-input formControlName="wifiPassword" type="text"></ion-input>
</ion-item>

<ion-item>
  <ion-label position="stacked">Phone Number</ion-
label>
  <ion-input formControlName="phoneNumber" type="text"></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 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" (input)="saveForm()">
```

We define the `formGroup` property as `campDetailsForm` which we will use with the Form Builder very shortly. We also listen for the `(input)` event and trigger the `saveForm` function when it is detected, this means that the `saveForm` function will trigger every time a user changes a specific input. Usually, on a form, you would instead listen for the `(submit)` event and handle the data then, but we don't want the user to have to hit a 'Save' button or anything like that, we want it to just save as soon as they enter a new value.

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 Form Builder 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(private 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 Form Builder group to it here. To create the group, we supply all of the **formControlName** names 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 doing something like this:

```
gateAccessCode: ['', Validators.required]
```

this will make the `gateAccessCode` field a required field. That's all there is to set up our form, now we just need to implement the `saveForm` function.

> **Modify the `saveForm` function in `src/app/camp-details/camp-details.page.ts` to reflect the following:**

```
saveForm(): void {  
  
    //this.dataService.setCampDetails(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 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 of course).

Remember, the `saveForm` function gets triggered every time the user makes a change to any input field, so whenever they make a change we read the values and send them off for saving.

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" (input)="saveForm()">

    <ion-item>
      <ion-label position="stacked">Car Registration</ion-
label>
      <ion-input formControlName="carRegistration"
type="text"></ion-input>
    </ion-item>

    <ion-item>
      <ion-label position="stacked">Trailer
Registration</ion-label>
      <ion-input formControlName="trailerRegistration"
type="text"></ion-input>
    </ion-item>
```

```

<ion-item>
    <ion-label position="stacked">Trailer
Dimensions</ion-label>
    <ion-input formControlName="trailerDimensions"
type="text"></ion-input>
</ion-item>

<ion-item>
    <ion-label position="stacked">Phone Number</ion-
label>
    <ion-input formControlName="phoneNumber" type="text">
</ion-input>
</ion-item>

<ion-item>
    <ion-label position="stacked">Notes</ion-label>
    <ion-textarea 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(private formBuilder: FormBuilder, private
dataService: DataService) {

  this.myDetailsForm = formBuilder.group({
    carRegistration: [''],
    trailerRegistration: [''],
    trailerDimensions: [''],
    phoneNumber: [''],
    notes: ['']
  });

}
```

> **Modify the saveForm function in src/app/my-details/my-details.page.ts to reflect the following:**

```
saveForm(): void {
  //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 Form Builder 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's 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 a custom Google Maps component that we generated earlier. This will also make it easier to reuse the same code in another application 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 their 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.

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, Input, Renderer2, ElementRef, Inject } from '@angular/core';
import { Platform } from '@ionic/angular';
import { DOCUMENT } from '@angular/common';
import { Plugins } from '@capacitor/core';

const { Geolocation, Network } = Plugins;

declare var google;
```

```
@Component({
  selector: 'google-map',
  templateUrl: './google-map.component.html',
  styleUrls: ['./google-map.component.scss']
})
export class GoogleMapComponent {

  @Input('apiKey') apiKey: string;

  public map: any;
  public marker: any;
  public firstLoadFailed: boolean = false;
  private mapsLoaded: boolean = false;
  private networkHandler = null;
  public connectionAvailable = true;

  constructor(
    private renderer: Renderer2,
    private element: ElementRef,
    private platform: Platform,
    @Inject(DOCUMENT) private _document){

  }

  public init(): Promise<any> {
    return Promise.resolve(true);
  }
}
```

```
private loadSDK(): Promise<any> {
    return Promise.resolve(true);
}

private injectSDK(): Promise<any> {
    return Promise.resolve(true);
}

private initMap(): Promise<any> {
    return Promise.resolve(true);
}

disableMap(): void {

}

enableMap(): void {

}

addConnectivityListeners(): void {

}

public changeMarker(lat: number, lng: number): 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 send the apiKey as input to the component. We grab this value using @Input
- 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

We also have a bunch of member variables set up:

- map - This will hold a reference to the Google Map
- marker - This will hold a reference to the marker added to the map
- firstLoadFailed - This indicates whether or not our first attempt to load the map manually failed
- mapsLoaded - This is a flag to keep track of whether or not the maps have been loaded
- networkHandler - This is a reference to the handler that keeps track of the online/offline status
- connectionAvailable - This is a flag that indicates whether or not a connection is available

Then we have our functions. The `init` function is what we will call from our location page to initialise the map, most of the rest of the functions are involved in the process of injecting the Google Maps SDK and initialising the map. We are going to work through implementing these one at a time.

> **Modify the `init` function to reflect the following:**

```
public init(): Promise<any> {

    return new Promise((resolve, reject) => {

        if(typeof(google) == 'undefined'){

            this.loadSDK().then((res) => {

                this.initMap().then((res) => {
                    this.enableMap();
                    resolve(true);
                }, (err) => {
                    this.disableMap();
                    reject(err);
                });
            });

            }, (err) => {

                this.firstLoadFailed = true;
                reject(err);
            });
    });
}
```

```
    });

} else {
    reject('Google maps already initialised');
}

});

}
```

First, notice that this function is public and that it returns a promise. We want to call this function from our location page, so we make it public (meaning it can be accessed from outside of this class). We also want to know when the whole process has finished loading the maps and initialising it, so by returning a promise we can let our location page know when it is ready.

The function first checks if google is already defined (we don't want to go through the process of injecting the SDK twice), and if it isn't it makes a call to the loadSDK function which we are about to implement. If the SDK is successfully loaded we call enableMap which we will implement later, and if it fails we call disableMap which we will also implement later.

Another important note is that we keep track of if the first load failed. We will set up some connectivity listeners to automatically attempt to run this process when the user is online, but we want a chance to trigger it manually first. We only want it to try to automatically

connect after we have tried to manually initiate the process. We will use this flag later.

> **Modify the LoadSDK function to reflect the following:**

```
private loadSDK(): Promise<any> {

    console.log('Loading Google Maps SDK');
    this.addConnectivityListeners();

    return new Promise((resolve, reject) => {

        if(!this.mapsLoaded){

            Network.getStatus().then((status) => {

                if(status.connected){

                    this.injectSDK().then((res) => {
                        resolve(true);
                    }, (err) => {
                        reject(err);
                    });
                } else {
                    reject('Not online');
                }
            })
        }
    })
}
```

```
}, (err) => {

    // NOTE: navigator.onLine temporarily required
    // until Network Capacitor plugin has web implementation

    if(navigator.onLine){

        this.injectSDK().then((res) => {
            resolve(true);
        }, (err) => {
            reject(err);
        });
    } else {
        reject('Not online');
    }

}).catch((err) => { console.warn(err); });

} else {
    reject('SDK already loaded');
}

});

}
```

The purpose of this function is to start the process of loading the SDK and to determine what exactly should be done based on the user's current connectivity status. In short, if the user is connected to the Internet we call the `injectSDK` function, but if they are not, then we don't. We have to do this twice here because we are using the Network Capacitor plugin to check the network status, but this does not currently have a web implementation (it will only work on iOS and Android). Since we want to be able to test this through the browser, we add a fallback for detecting the network status.

Also, notice that at the start of the function we call the `addConnectivityListeners` function. This will handle automatically loading the SDK when an Internet connection is available after our initial attempt.

> **Modify the `injectSDK` function to reflect the following:**

```
private injectSDK(): Promise<any> {  
  
    return new Promise((resolve, reject) => {  
  
        window['mapInit'] = () => {  
            this.mapsLoaded = true;  
            resolve(true);  
        }  
  
        let script = this.renderer.createElement('script');  
        script.id = 'googleMaps';  
    }  
}
```

```

        if(this.apiKey){
            script.src =
'https://maps.googleapis.com/maps/api/js?key=' + this.apiKey +
'&callback=mapInit';
        } else {
            script.src =
'https://maps.googleapis.com/maps/api/js?callback=mapInit';
        }

        this.renderer.appendChild(this._document.body,
script);
    });

}

```

This is the part where we actually inject the 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. It is not typical to define anything 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.

> **Modify the initMap function to reflect the following:**

```
private initMap(): Promise<any> {

    return new Promise((resolve, reject) => {

        Geolocation.getCurrentPosition({enableHighAccuracy:
            true, timeout: 10000}).then((position) => {

            console.log(position);

            let latLng = new
                google.maps.LatLng(position.coords.latitude,
                    position.coords.longitude);

            let mapOptions = {
                center: latLng,
                zoom: 15
            };

            this.map = new
                google.maps.Map(this.element.nativeElement, mapOptions);
            resolve(true);

        }, (err) => {
            console.log(err);
            reject('Could not initialise map');
        });
    });
}
```

```
});  
  
}  
  
}
```

This function is called once the SDK has finished loading, and it is responsible for setting up our map. We want to center the map on the user's location, so we use the Geolocation API from Capacitor to grab the users location. We then use that location in the options object for google maps and create a new map using google.maps.Map. We supply this with the native element of our component (which we already set up a reference to by using ElementRef) and our map options.

> Modify the disableMap and enableMap functions to reflect the following:

```
disableMap(): void {  
    this.connectionAvailable = false;  
}  
  
enableMap(): void {  
    this.connectionAvailable = true;  
}
```

These functions just toggle our connectionAvailable flag. We will use this flag to modify the styling of our component to indicate the current status to the user.

> Modify the `addConnectivityListeners` function to reflect the following:

```
addConnectivityListeners(): void {

    console.warn('The Capacitor Network API does not currently
have a web implementation. This will only work when running as an
iOS/Android app');

    if(this.platform.is('cordova')){

        this.networkHandler =
Network.addListener('networkStatusChange', (status) => {

            if(status.connected){

                if(typeof google == 'undefined' &&
this.firstLoadFailed){

                    this.init().then((res) => {
                        console.log('Google Maps ready.')
                    }, (err) => {
                        console.log(err);
                    });
                }
            } else {
                this.enableMap();
            }
        });
    }
}
```

```
    } else {

        this.disableMap();
    }

});

}

}
```

This is the function that I mentioned would handle automatically injecting the SDK if the first attempt failed. We do this by adding a listener for `networkStatusChange` which will trigger each time the network status changes. If we detect that the user is online and that `google` has not yet been defined, we attempt to call the `init` function again to kick off the loading process.

If we detect that the user is now offline, then we trigger the `disableMap` function to disable the map.

> **Modify the `changeMarker` function to reflect the following:**

```
public changeMarker(lat: number, lng: number): void {
```

```

let latLng = new google.maps.LatLng(lat, lng);

let marker = new google.maps.Marker({
  map: this.map,
  animation: google.maps.Animation.DROP,
  position: latLng
});

// Remove existing marker if it exists
if(this.marker){
  this.marker.setMap(null);
}

// Add new marker
this.marker = marker;

}

```

The core functions for loading and displaying the map have already been completed, this function is a bit of an extra utility function. We will be able to call this function from our location page to add a marker to the map at the specified latitude and longitude.

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></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: #e7e7e7;
```

```

}

#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;
  position: relative;
  font-size: 1.6em;
  top: 30%;
}

```

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>
  <google-map apiKey="YOUR-API-KEY"></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, someone could potentially find and use your API key without your knowledge and add to your usage.

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) map: GoogleMapComponent;
```

Then we are going to make a call to the `init` function of the component inside of the `ngOnInit` lifecycle hook that runs when our location page is loaded.

> Modify the `ngOnInit` function in `src/app/location/location.page.ts` to the following:

```
ngOnInit(){

    this.map.init().then((res) => {

        console.log('map ready!');

    }, (err) => {
        console.log(err);
    });
}
```

```
}
```

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!

Next, we are going to implement the `setLocation` function, which will set the user's camp location to their current location.

> **Modify the `setLocation` function in `src/app/location/location.page.ts` to reflect the following:**

```
setLocation(): void {  
  
  this.loadingCtrl.create({  
    message: 'Setting current location...'  
  }).then((overlay) => {  
  
    overlay.present();  
  
    Geolocation.getCurrentPosition().then((position) => {  
  
      overlay.dismiss();  
  
      this.latitude = position.coords.latitude;  
      this.longitude = position.coords.longitude;  
    })  
  })  
}
```

```
this.map.changeMarker(this.latitude, this.longitude);

let data = {
  latitude: this.latitude,
  longitude: this.longitude
};

//this.dataService.setLocation(data);

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'
    }
  ]
}).then((alert) => {
  alert.present();
});

}, (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 `this.latitude` and `this.longitude` values 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. Now we just have one more function to define:

> Modify the `takeMeHome` function in `src/app/location/location.ts` to reflect the following:

```
takeMeHome(): void {
```

```

if(!this.latitude || !this.longitude){

    this.alertCtrl.create({
        header: 'Nowhere to go!',
        message: 'You need to set your camp location first.',
        buttons: [
            {
                text: 'Ok'
            }
        ]
    }).then((alert) => {
        alert.present();
    });

} else {

    let destination = this.latitude + ',' + this.longitude;

    if(this.platform.is('ios')){
        window.open('maps://?q=' + destination, '_system');
    } else {
        let 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.latitude` and `this.longitude` values are set, and if they are 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.

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 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';

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

```
constructor(private storage: Storage) {  
  
}  
  
setMyDetails(data): void {  
    this.storage.set('mydetails', data);  
}  
  
setCampDetails(data): void {  
    this.storage.set('campdetails', data);  
}  
  
setLocation(data): void {  
    this.storage.set('location', data);  
}  
  
getMyDetails(): Promise<any> {  
    return this.storage.get('mydetails');  
}  
  
getCampDetails(): Promise<any> {  
    return this.storage.get('campdetails');  
}  
  
getLocation(): Promise<any> {  
    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` function in `src/app/location/location.page.ts` to reflect the following:**

```
ngOnInit(){

    this.dataService.getLocation().then((location) => {

        this.map.init().then((res) => {

            if(location != null){

                this.latitude = location.latitude;
                this.longitude = location.longitude;

                this.map.changeMarker(this.latitude, this.longitude);

            }

        }, (err) => {
            console.log(err);
        });
    });

}
```

Rather than just directly calling the `init` function, we first grab the users saved location from the data service (which we are yet to implement). We then call the `init` function, and once the `init` process has finished we check if there was a location saved. If there was a saved location, then we add a marker to the map at that position.

Next, let's take care of loading in the form data for our Camp Details page.

> Add an `ngOnInit` function to `src/app/camp-details/camp-details.page.ts` that reflects the following:

```
ngOnInit(){

    this.dataService.getCampDetails().then((details) => {

        let formControls: any = this.campDetailsForm.controls;

        if(details != null){

            formControls.gateAccessCode.setValue(details.gateAccessCode);

            formControls.ammenitiesCode.setValue(details.ammenitiesCode);

            formControls.wifiPassword.setValue(details.wifiPassword);

            formControls.phoneNumber.setValue(details.phoneNumber);
    
```

```
    formControls.departure.setValue(details.departure);
    formControls.notes.setValue(details.notes);
}

});

}
```

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 Form Builder, 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. So, instead we create the Form Builder 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 `saveForm` method previously, so let's rectify that now.

> Modify the `saveForm` function in `src/app/camp-details/camp-details.page.ts` to reflect the following:

```
saveForm(): void {  
  this.dataService.setCampDetails(this.campDetailsForm.value);  
}
```

Now we just need to do the same for the My Details page.

> Add an `ngOnInit` function to `src/app/my-details/my-details.page.ts` that reflects the following:

```
ngOnInit() {  
  
  this.dataService.getMyDetails().then((details) => {  
  
    let formControls: any = this.myDetailsForm.controls;  
  
    if(details != null){  
  
      formControls.carRegistration.setValue(details.carRegistration);  
  
      formControls.trailerRegistration.setValue(details.trailerRegistration);  
    }  
  })  
}
```

```
formControls.trailerDimensions.setValue(details.trailerDimensions);

    formControls.phoneNumber.setValue(details.phoneNumber);
    formControls.notes.setValue(details.notes);

}

});

}
```

> **Modify the saveForm function in src/app/my-details/my-details.page.ts to reflect the following:**

```
saveForm(): void {
    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() function in src/app/location/location.page.ts:**

```
setLocation(): void {

    this.loadingCtrl.create({
        message: 'Setting current location...'
    }).then((overlay) => {

        overlay.present();

        Geolocation.getCurrentPosition().then((position) => {

            overlay.dismiss();

            this.latitude = position.coords.latitude;
            this.longitude = position.coords.longitude;

            this.map.changeMarker(this.latitude, this.longitude);

            let data = {
                latitude: this.latitude,
                longitude: this.longitude
            };

            this.dataService.setLocation(data);

            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'
        }
    ]
}).then((alert) => {
    alert.present();
});

}, (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: #5b91da;
```

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: 39px;  
  margin-top: 6px;  
}  
  
ion-item {
```

```
--background-active: #fff;  
}  
  
ion-input, ion-textarea {  
  background-color: #F3F3F3;  
  border: 1px solid #cecece;  
  padding-left: 10px;  
}  
  
ion-textarea {  
  height: 200px;  
}  
  
textarea {  
  height: 180px;  
}
```

The main thing we are trying to achieve here is to add a bit of styling to our input fields. Before doing this, the input fields were white and so was the background, so you couldn't even see where the input fields were. Now they will have a grey background colour and a bit of extra styling. We've also added a bit of styling to the textarea to expand its default height.

The application should now look like this:

My Details

Update this form with your details so you have an easy reference for later.

Car Registration

IAMCOOL

Trailer Registration

TRAILER

Trailer Dimensions

8 x 16ft

Phone Number

555444333

Notes



Location



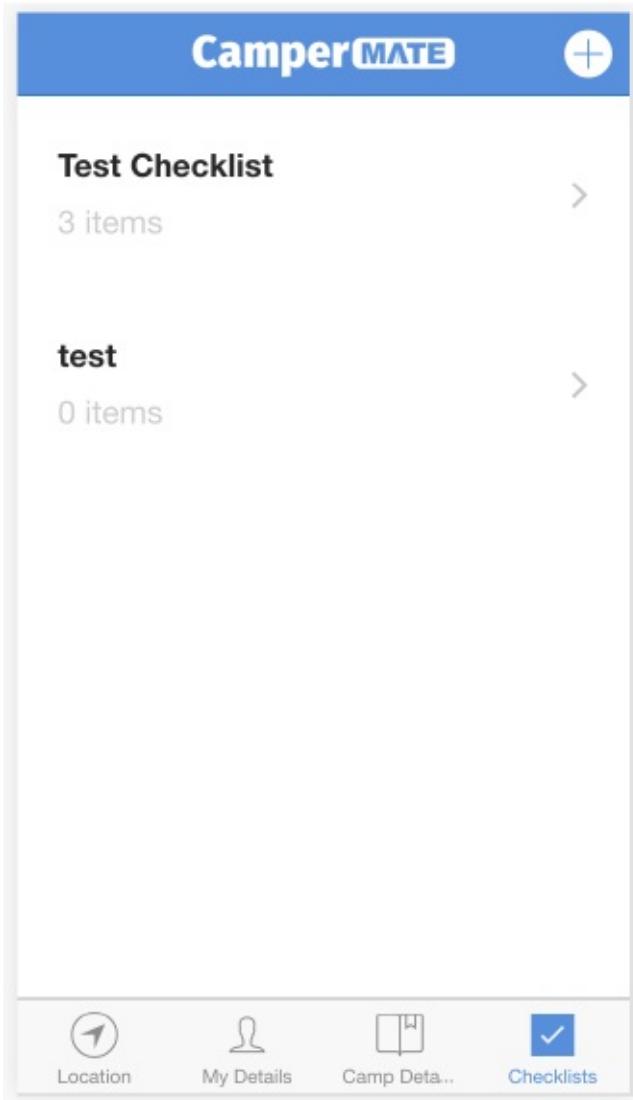
My Details



Camp Data...



Checklists



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
- 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]**
- Use local notifications (see the Snapaday application for help) to notify the user the day before their departure date **[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 cranked up a notch 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 all things caravan and camping with anybody else who is using the application. The coolest thing about this application is the integration with **PouchDB** for storing local data and **Cloudant** for syncing that PouchDB data to a remote backend. This means that the data can be available to users when they are offline, and when they come back online again the latest updates will be fetched from the remote backend.

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 a bunch of different things including:

- Navigation
- Using a Sliding Menu
- Using PouchDB to store local data
- Using Cloudant to store remote data
- Enabling online/offline sync
- Simple Animation
- Using the Facebook API for authentication and other features
- Configuring native projects for plugin integrations
- Protecting routes with an "Auth Guard" service
- Updating and displaying data in real time

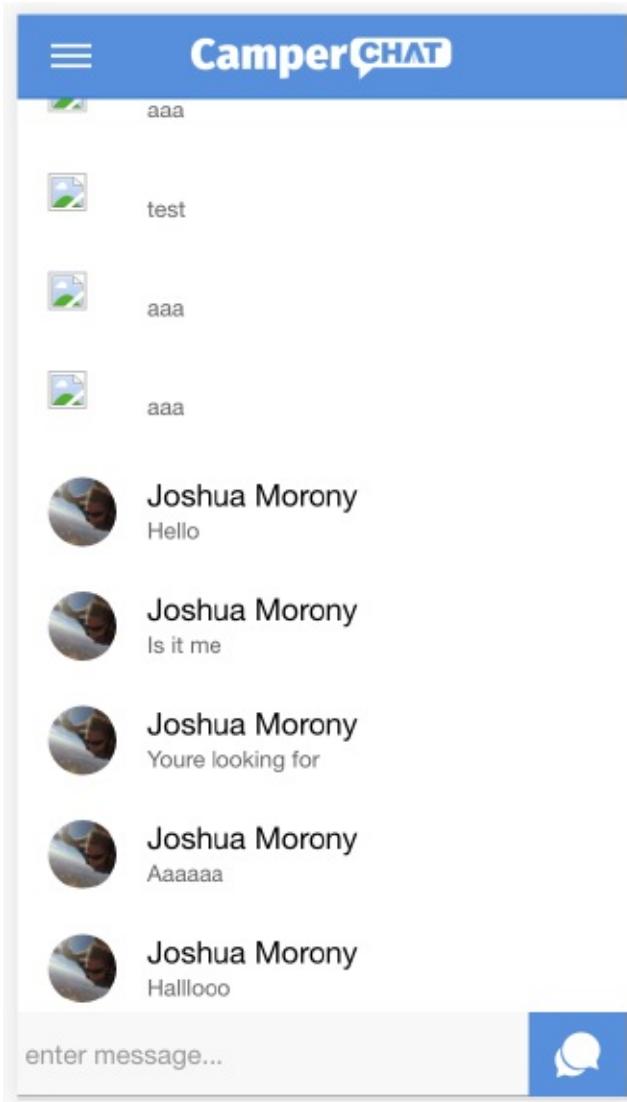
and here are some screenshots to help you get an idea of what the application will look like:

Camper **CHAT**

Camper Chat allows you to talk to other **caravaners, RV'ers, roadtrippers 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.

Log in with **Facebook** below to get started.

Connect with Facebook



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. For example, although we are using Facebook for authentication in the application, this is primarily to gain access to the user's data. Logging in with Facebook is required to access the application, but the database that we will be interacting with will be public (as the API key for the database is embedded in the client side code, meaning anybody *could* access that key if they wanted to). The controls in place to keep people from accessing the logged in portions of the application are also only enforced on the client side, so anybody determined enough could bypass it.

If you wanted to build a private chat application where authentication is required (not just encouraged), you would need to add additional security measures to the application (e.g. creating a proxy server to handle communication with the database, so that the API key is not exposed in your client-side code). This topic is too large for this book, but if you are interested in learning more you will find articles about this on my blog.

Lesson Structure

1. [Getting Ready](#)
2. [Login Page and Sliding Menu Layout](#)
3. [Using Facebook for Authentication](#)
4. [Creating and Displaying Messages & Navigation](#)
5. [Creating a Local and Remote Backend with PouchDB and Cloudant](#)
6. [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
```

or

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

before you continue. If you run into any trouble installing Ionic or generating new projects, make sure that you have the most recent [NodeJS LTS](#) version installed. After you have that installed, you should also run the following command:

```
npm uninstall -g ionic
```

before attempting to install again.

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 camperchat blank --type=angular
```

When asked, **do not** integrate the Ionic Appflow SDK.

> Make the new project your current working directory by running the following command:

```
cd camperchat
```

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:

Ionic Blank

The world is your oyster.

If you get lost, the [docs](#) will be your guide.

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 a data 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.

> Run the following command to generate a Data service:

```
ionic g service services/Data
```

NOTE: Notice that we use services/Data instead of just Data 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 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 { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', loadChildren:
    './home/home.module#HomePageModule' },
  { path: 'login', loadChildren:
    './login/login.module#LoginPageModule' },
  { path: 'about', loadChildren:
    './about/about.module#AboutPageModule' },
];

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

Install PouchDB

We will be making use of an external library called **PouchDB** in this application. In order to use it, we will need to install it:

```
npm install pouchdb --save
```

We will also need to install "typings" for PouchDB so that the TypeScript compiler doesn't complain (since it doesn't know what PouchDB is).

> **Run the following command to install types for PouchDB**

```
npm install @types/pouchdb --save
```

Configure Capacitor

Capacitor is not included by default in our applications, so we need to set it up. You should still complete this step even if you don't intend to build for iOS and Android because we will also be using Capacitor for the web platform.

To set up Capacitor in an Ionic application, all you need to do is run a single command.

> **Run the following command to enable Capacitor:**

```
ionic integrations enable capacitor
```

This will handle setting up the Capacitor CLI and Capacitor Library in your project.

Keep in mind that this will give your application a default 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 now. So, you may want to open the **capacitor.config.json** file in your project and change `appId` to something like `com.yourcompany.yourproject`.

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. 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
```

```
ionic cap add android
```

Once you have added the platforms you want, you can copy over your application code and dependencies over 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.

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'
```

```
)  
  
export class AppComponent {  
  
  constructor() {  
  
    SplashScreen.hide().catch((err) => {  
      console.warn(err);  
    });  
  
    StatusBar.hide().catch((err) => {  
      console.warn(err);  
    });  
  
  }  
  
}  
  
}
```

All we are doing here is hiding the StatusBar but you can use this API to style the Status Bar however you like. The status bar is the bar at the top of native applications that display the time, battery level, etc.

We also have a check here to ensure that the application is running on iOS or Android before making calls to the plugins, as SplashScreen and StatusBar do not have web implementations. This is just a temporary measure, as eventually all Capacitor plugins will work either on the web and natively, or they will fail gracefully on the web where the functionality is not available.

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 likely to get wiped by the operating system).

To enable Ionic storage, you need to install it.

> **Install Ionic Storage with the following command:**

```
npm install @ionic/storage --save
```

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 { RouterModule, RouteReuseStrategy, Routes } 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 --save
```

We are installing the SQLite plugin in the application so that the Ionic Storage API can make use of native storage.

We will also be making use of the Facebook plugin and its Ionic Native wrapper.

> Run the following commands to add the Facebook plugin:

```
npm install cordova-plugin-facebook4 --save
```

```
npm install --save @ionic-native/facebook@beta
```

At the moment, it is important to include the @beta tag as we need to use the latest version of the Ionic Native package. However, eventually you will be able to get rid of the @beta tag (if the install command is no longer working, you should remove it).

Since we are using Ionic Native, we will also need to add this plugin into your **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 { RouterModule, RouteReuseStrategy, Routes } 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(),
IonicStorageModule.forRoot(), AppRoutingModule],
  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 even 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 log in 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 padding>

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

  <ion-row class="login-description">
    <ion-col>
      <p>Camper Chat allows you to talk to other
      <strong>caravaners, RV'ers, road trippers and campers</strong>
    </ion-col>
  </ion-row>
```

near you. Use Camper Chat to give and receive tips, ask for help with your flat tyre, or just have a friendly chat.</p>

```
<p>Log in with <strong>Facebook</strong> below to get started.</p>
```

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

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

```
<ion-row>
```

```
<ion-col>
```

```
<a (click)="login()"></a>
```

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

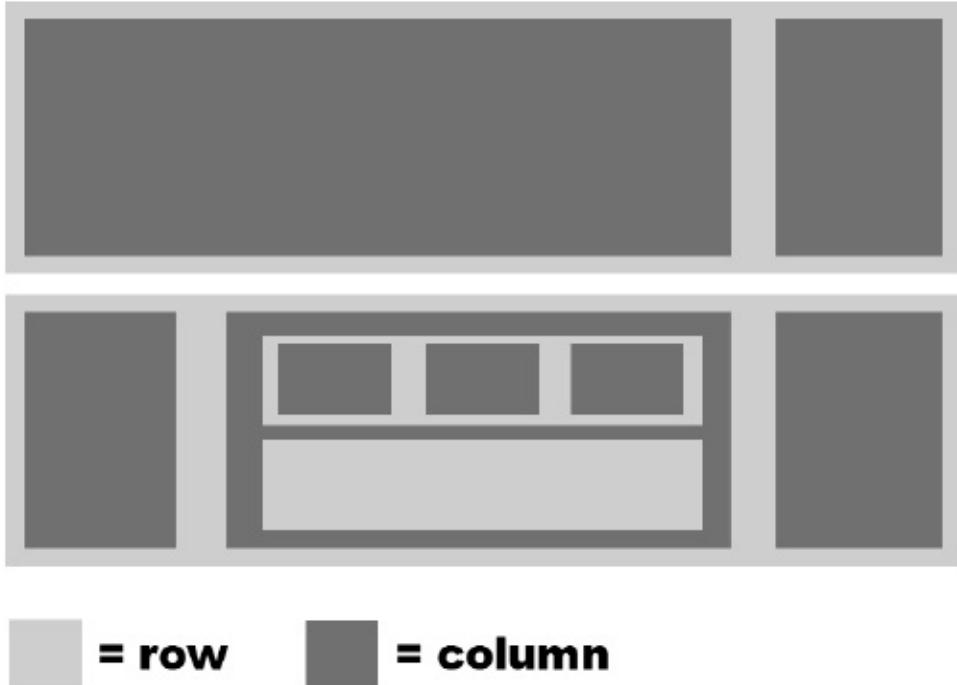
```
</ion-row>
```

```
</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 usualy `<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're not doing anything too crazy here, but we are using Ionic's grid system, which consists of columns and rows, to set up our layout. It's a pretty basic layout, but it is a good chance to get familiar with how the grid system works. When using the grid system

with `<ion-row>` and `<ion-col>`, **rows** get placed underneath one another and **cols** within those rows appear side by side. This diagram should help illustrate how that works:



This is a very simple example because we are just using the grid to display three sections underneath each other, but you can also include multiple columns within rows and even define how wide they are like this:

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

The layout is 12 columns wide in total, so a column width of 6 will take up half of the

available space.

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, AlertController,
NavController } from '@ionic/angular';
import { Facebook } from '@ionic-native/facebook/ngx';
import { DataService } from '../services/data.service';

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

  private loading;

  constructor(
```

```
    private facebook: Facebook,  
    private menu: MenuController,  
    private navCtrl: NavController,  
    private loadingCtrl: LoadingController,  
    private alertCtrl: AlertController,  
    private dataService: DataService  
) {  
  
}  
  
ngOnInit(){  
  
    this.menu.enable(false);  
  
}  
  
login(): void {  
  
    this.getProfile();  
  
}  
  
getProfile(): void {  
  
    this.menu.enable(true);  
    this.navCtrl.navigateRoot('/home');  
  
}
```

```
}
```

As you can see in the code above, we've injected and set up our NavController and we're using it to navigate to the HomePage when the login function is called. This code is being run through the `getProfile()` function because later we will be grabbing some information about the user from Facebook after they have authenticated, but before we send them to the next page. Since we haven't implemented that yet we just want the code to flow right through to the success scenario.

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.

We've also imported the Facebook plugin from Ionic Native which we will use later. Remember, it is possible to access and use plugins without Ionic Native, but Ionic Native can make the plugin a little nicer to work with.

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, NavController } from '@ionic/angular';
import { Facebook } from '@ionic-native/facebook/ngx';
import { DataService } from './services/data.service';

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {

  constructor(
    private navCtrl: NavController,
    private menu: MenuController,
    private facebook: Facebook,
    private dataService: DataService) {

    SplashScreen.hide().catch((err) => {
      console.warn(err);
    });

    StatusBar.hide().catch((err) => {
      console.warn(err);
    });
  }
}
```

```
}

logout(): void {
}

}
```

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 Data service, which we will use for saving and accessing data later, the Menu Controller, which will allow us to interact with the sliding menu we are creating. We've also imported Facebook from Ionic Native so that the logout function can interact with the Facebook API.

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 side="start">
  <ion-header>
    <ion-toolbar color="primary">
      <ion-title>Menu</ion-title>
```

```
</ion-toolbar>

</ion-header>

<ion-content>

  <ion-list>

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

Except for one bit, and that's that we have added a `main` attribute. In order to behave correctly, the menu needs to know what content to attach itself to. So, 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>
  <ion-toolbar color="primary">
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title><img src ="assets/images/logo.png" class="logo" />
  </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">
        <strong>Name here</strong>
        <p>message here</p>
      </div>
    </ion-item>

  </ion-list>
```

```
</ion-content>

<ion-footer>

  <ion-toolbar>

    <textarea [(ngModel)]="chatMessage" spellcheck="true"
autoComplete="true" autocorrect="true" rows="1" class="chat-
input" placeholder="type message...">
    </textarea>

    <ion-buttons slot="primary">
      <ion-button (click)="sendMessage()">
        <ion-icon slot="icon-only" name="send"></ion-icon>
      </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 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, and we've added a bit of inline styling to make it float to the right side of the toolbar.

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 } from '@ionic/angular';
import { DataService } from '../services/data.service';

@Component({
  selector: 'app-page-home',
  templateUrl: './home.page.html',
```

```

    styleUrls: ['./home.page.scss']
  })

export class HomePage implements OnInit {

  @ViewChild(IonContent) contentArea: IonContent;
  @ViewChild(IonList, {read: ElementRef}) chatList: ElementRef;

  public chatMessage: string = '';
  private mutationObserver: MutationObserver;

  constructor(public dataService: DataService){

  }

  ngOnInit(){

  }

  sendMessage(): void {
  }

}

}

```

Mostly pretty simple stuff here. We're importing and injecting our data service again and we set up the `this.chatMessage` class member that our input field from the template links 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 making use of both of these references later, but we don't need to worry about that just yet.

If you run your application using:

```
ionic serve
```

you should have two pages now that look like this:



CamperCHAT

Name here

message here

enter message...



Camper Chat allows you to talk to other **caravaners, RV'ers, roadtrippers 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.

Log in with **Facebook** below to get started.

Connect with Facebook

As is usually the case, it looks pretty ugly right now but the basic structure for the layout we need is there. We have a login page with a login option, 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 juicy data to use in the application like the user's name and profile picture).

Lesson 4: Facebook Authentication

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](#).

Using social authentication like the Facebook API provides 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.

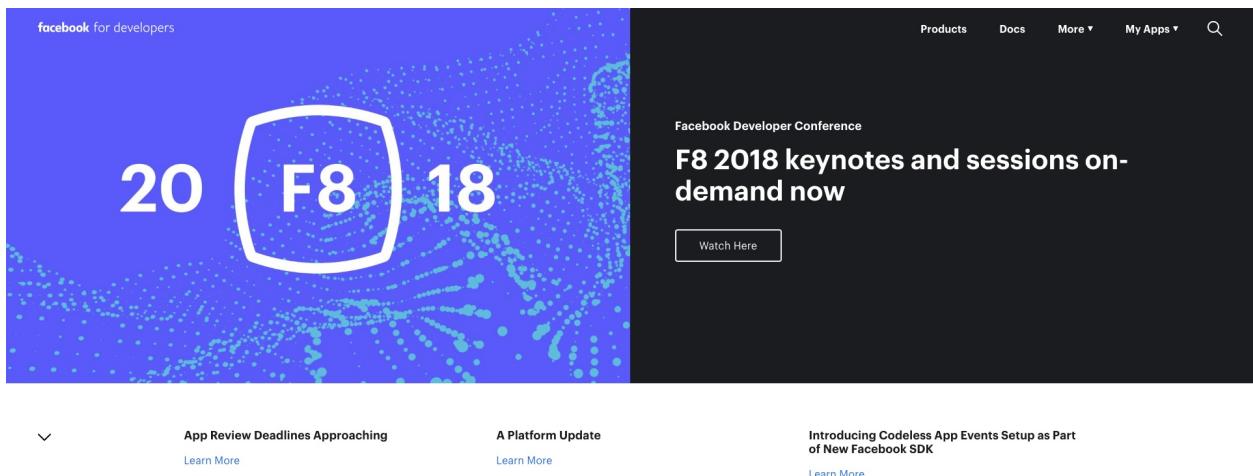
Let's walk through how to set up the Facebook plugin and then we can look at integrating into our application.

IMPORTANT: The Facebook connect plugin will only work when running on a real mobile device, if you try to run it through your desktop browser you will just receive errors. Instead of testing by running `ionic serve`, you should instead run it on a real device. 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.

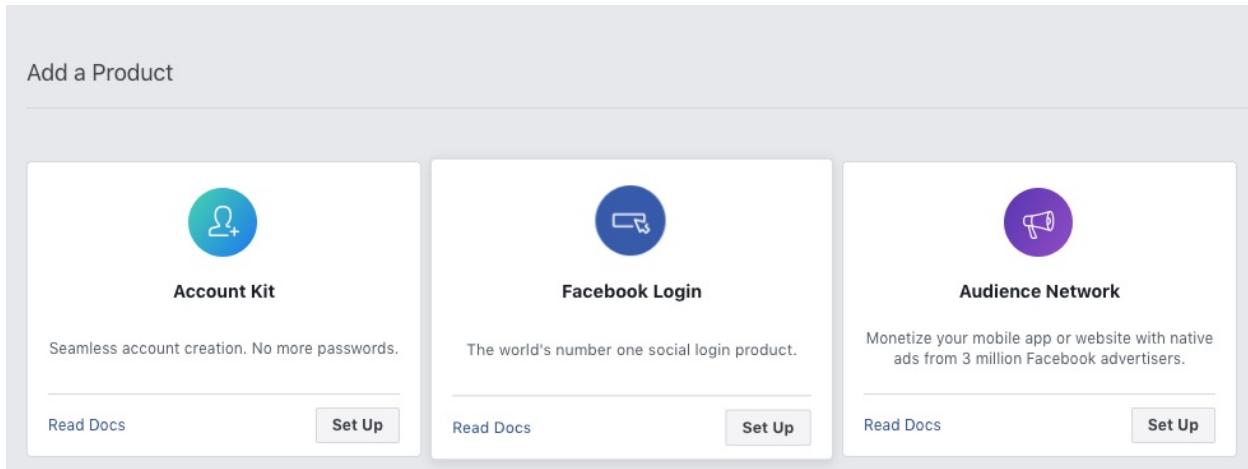
Setting up a Facebook App

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, though. We are going to need to supply the plugin with an **App Name** and an **App ID**. To do that, we need to create an application on Facebook's developer's platform, so let's walk through doing that.

First, you will need to go to developers.facebook.com and sign up if you haven't already. Once you've done that you should see a screen like this:



Under **My Apps** in the top-right click **Add New App**. Choose a name for the application and then click **Create App ID**. You will then be taken to the Dashboard for your new Facebook application, and you will see a variety of options:



We want to set up **Facebook Login**, so click **Set Up** on that option. You will then be asked to choose a platform, but we actually want to set up multiple platforms. If you only intend to build your application for one platform, feel free to leave out the platform that you are not using.

Rather than going through Facebook's "Quickstart" guide, we are just going to jump straight into the configuration ourselves (because the way we need to implement it is a little different). Just click on the **Settings** option on the menu on the left and choose **Basic**.

On this screen, you will see the bit of information we are most interested in, the **App ID**. Make a note of this because we will be using it later. You will also need your **Display Name** or **App Name**, but that is just whatever you decided to call your application. There are many more fields that will need to be filled out before you can make your Facebook application live, but we can skip those for now.

If you scroll to the bottom of this settings page, you will see an option to add a platform. If you are targeting both iOS and Android, you will need to add a platform for each of these.

Set up the Platforms

To set up a platform for iOS, first, click **Add Platform**, and then choose the **iOS** option.

You will then need to fill out the details to reflect the following:

The screenshot shows the 'ios' configuration screen for a Facebook app. It includes fields for 'Bundle ID' (set to 'com.joshmorony.camperchat'), 'iPhone Store ID' (placeholder 'The ID to identify your app in the iOS Store'), 'URL Scheme Suffix (Optional)' (empty), 'iPad Store ID' (placeholder 'The ID to identify your app in the iPad Store'), 'Single Sign On' (set to 'Yes'), 'Deep Linking' (set to 'No'), and a section for 'Log In-App Purchase Events Automatically (Recommended)' which is turned off ('No'). A note explains that turning it on logs all in-app purchase events automatically on iOS, requiring version 3.22 of the Facebook SDK or newer.

You should replace **Bundle ID** with your applications own Bundle ID (this is the value you used when setting up Capacitor, and it is contained in the **capacitor.config.json** file). To set up the **Android** platform you should do the same, except you should configure the options as follows:

The screenshot shows the 'Android' configuration screen for a Facebook app. It includes fields for 'Google Play Package Name' (set to 'com.joshmorony.camperchat'), 'Class Name' (placeholder 'The Main Activity you want Facebook to launch'), 'Key Hashes' (containing '+MJKOYOURKEYHASHGe9NGs='), 'Amazon Appstore URL (Optional)' (placeholder 'Ex. http://www.amazon.com/dp/B004GJDQT8'), 'Single Sign On' (set to 'Yes'), 'Deep Linking' (set to 'No'), and a section for 'Log In-App Purchase Events Automatically (Recommended)' which is turned off ('No'). A note explains that turning it on logs all in-app purchase events automatically on Android, requiring version 4.27 of the Android Facebook SDK or newer.

This is basically the same idea, except that a **Key Hash** needs to be supplied. This is a "hash" of the "keystore" file used to sign your Android application. There are more in-depth instructions available in the [Building for Android and Distributing to Google Play](#) lesson, but if you want to just generate a hash for the default debug keystore that Android Studio uses, you and 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. Once you do this, your key hash will be displayed on the screen. Keep in mind that you will need to update this key hash later to reflect the keystore file that is used to sign the production version of your application.

This is all we need to do in the Facebook interface for now, but keep in mind that when your app is ready you will need to make it "live" in the Facebook Developers dashboard.

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. 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:

```
app > resources > values > strings.xml
```

Inside of this file, you will need to add the following entries **inside** of the <resources> tag:

```
<string name="fb_app_id">YOUR APP ID</string>
<string name="fb_app_name">YOUR APP NAME</string>
```

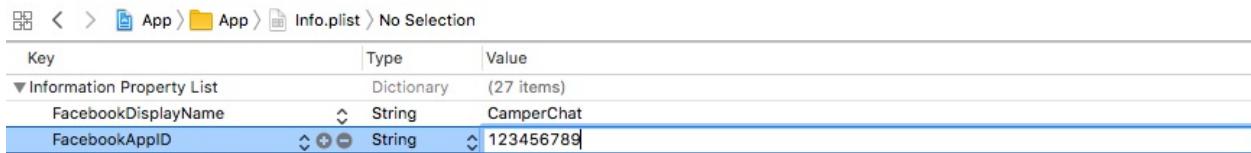
Make sure to replace **YOUR APP ID** with your Facebook **App ID** and **YOUR APP NAME** with your Facebook **App Name**. Now we need to do something similar for iOS. Open the iOS project with:

```
ionic cap open ios
```

Once the project is open, you should go to:

```
App > App > Info.plist
```

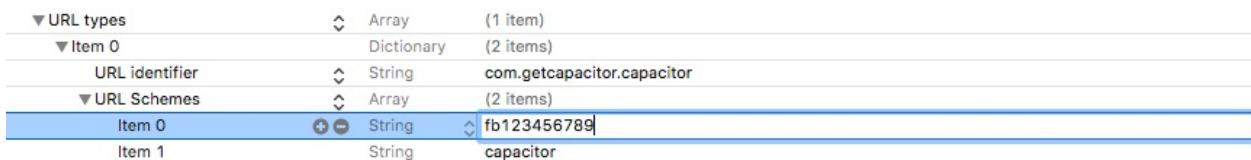
If you hover 'Information Property List' a little add icon will appear that you can use to add your **FacebookDisplayName** and **FacebookAppID**:



The screenshot shows the Xcode interface with the 'Info.plist' file open. The 'FacebookAppID' key is selected, and its value '123456789' is displayed in the text field. A blue selection bar is visible under the 'FacebookAppID' row.

Key	Type	Value
▼ Information Property List	Dictionary	(27 items)
FacebookDisplayName	String	CamperChat
FacebookAppID	String	123456789

There are also some additional steps specific to this Facebook plugin that we need to take, so let's do that whilst we are here. You will also need to add an entry under **URL Schemes** that reflects your Facebook App ID prefixed with **fb**, like this:



The screenshot shows the Xcode interface with the 'Info.plist' file open. The 'URL Schemes' array is selected, and its first item 'fb123456789' is displayed in the text field. A blue selection bar is visible under the 'Item 0' row.

▼ URL types	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
URL identifier	String	com.getcapacitor.capacitor
▼ URL Schemes	Array	(2 items)
Item 0	String	fb123456789
Item 1	String	capacitor

otherwise, you will get an error complaining about not having the URL scheme registered:



Finally, you will also need to add an array of values under **LSServiceSchemes** (make sure to switch this entry from a **String** to an **Array**):



The screenshot shows the Xcode interface with the 'Info.plist' file open. The 'LSServiceSchemes' array is selected, and its four items are listed: 'fbshareextension', 'fb-messenger-api', 'fbapi', and 'fbauth2'. A blue selection bar is visible under the 'Item 0' row.

▼ LSServiceSchemes	Array	(4 items)
Item 0	String	fbshareextension
Item 1	String	fb-messenger-api
Item 2	String	fbapi
Item 3	String	fbauth2

Setting up Authentication

We've got everything we need set up now, so we can jump into some more coding in our application. Integrating the Facebook API with Ionic Native is actually really easy, but before we do that we are going to set up our data service so that it can store some values from Facebook temporarily. The data service will primarily be used for handling storing and retrieving our message data later, but since we just need to store a couple of values from Facebook it's easy to just tack them on here.

> **Modify `src/app/services/data.service.ts` to reflect the following:**

```
import { Injectable } from '@angular/core';
import * as PouchDB from 'pouchdb/dist/pouchdb';

interface UserData {
  fbid: number;
  username: string;
  picture: string;
}

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

  public user: UserData = {
    fbid: null,
```

```
username: null,  
picture: null  
}  
  
constructor(){  
  
}  
  
}
```

We've set up a custom **interface** to enforce a particular data structure on our "user data" object. The object that represents the user's data should contain an fbid which is a number, username which is a string and a picture which is a string. We just initialise all of these values to null to begin with, as we will set them with the data we get back from Facebook later.

Now we're going to modify our login page to authenticate with Facebook when the user clicks the login button, rather than just going straight to the home page.

> **Modify the login function in src/app/pages/login/login.page.ts to reflect the following:**

```
login(): void {  
  
  this.loadingCtrl.create({
```

```
    message: 'Authenticating...',  
}).then((overlay) => {  
  
    this.loading = overlay;  
    this.loading.present();  
  
    this.facebook.login(['public_profile']).then((response) =>  
{  
  
        this.getProfile();  
  
    }, (err) => {  
  
        this.alertCtrl.create({  
            header: 'Oops!',  
            message: 'Something went wrong, please try again  
later.',  
            buttons: ['Ok']  
        }).then((alert) => {  
            this.loading.dismiss();  
            alert.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.

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", but there are other types of permissions you can request.

Take a look at [this page](#) for a full list. The `login` method returns a promise, so we set up a handler for the response and if it is successful we trigger the `getProfile` function we set up previously which will launch the home page. If it is not successful then we create and display an alert to the user.

We now know that a user has successfully authenticated with Facebook and that they should have access to our application, but we still need to find out a few details about them. In order for them to use the application we will also need their name and profile picture, so to grab this we are going to modify the `getProfile` function.

> **Modify the `getProfile` function in `src/app/pages/login/login.page.ts` to reflect the following:**

```
getProfile(): void {
```

```
    this.facebook.api('/me?fields=id,name,picture',
['public_profile']).then(
  (response) => {
    console.log(response);

    this.dataService.user.fbid = response.id;
    this.dataService.user.username = response.name;
    this.dataService.user.picture =
      response.picture.data.url;

    this.loading.dismiss().then(() => {
      this.menu.enable(true);
      this.navCtrl.navigateRoot('/home');
    });
  },
  (err) => {
    console.log(err);

    this.alertCtrl.create({
      header: 'Oops!',
      message: 'Something went wrong, please try again
later.',
```

```
        buttons: ['Ok']
      }).then((alert) => {
        this.loading.dismiss();
        alert.present();
      });

    }
  );
}

}
```

Now we are making a call to the `api` method that the Facebook plugin provides. This allows us to interact with [Facebooks Graph API](#) which is how you do just about everything with the Facebook API - you can get into some really complex stuff but we just want to use it to grab the users id, full name, and display picture.

The first parameter that needs to be provided to the `api` call is the endpoint you want to hit, which for us is `/me` because we want data about the currently logged in user, and we add a query string containing all the data fields we want returned. We also provide an array of permissions we require again and then handle the response.

If the response is successful we store those values we retrieved using our `dataService`, then we enable the sliding menu and switch to the home page. If the response is not successful then we again show the user an error message and keep them on the login page.

The only other bit of functionality we need to use from the Facebook API is the logout method. When a user logs out of our application we want to terminate that session with Facebook as well (this won't log them out of Facebook, it just means that they will need to reauthenticate with Facebook when using our application again). To do that we are going to modify our own logout function.

> **Modify the logout function in `src/app/app.component.ts` to reflect the following:**

```
logout(): void {  
  
    this.menu.close();  
    this.menu.enable(false);  
  
    this.dataService.user.fbid = null;  
    this.dataService.user.username = null;  
    this.dataService.user.picture = null;  
  
    this.facebook.logout();  
    this.navCtrl.navigateRoot('/login');  
  
}
```

First, we handle stuff on our end. We close the menu and disable it, and then switch back to the login page. We reset all the Facebook data that was stored in our data service and then we call the logout method on the Facebook API. The user would now be back on the login page in our application and unauthenticated.

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 for this particular app since it won't be used on the web, and the URL bar isn't easily accessed, but we are going to walk through a way to prevent this anyway.

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 sensible path within your application. If certain pages in your application contain sensitive 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.

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, CanActivate } from '@angular/router';
import { DataService } from './data.service';
```

```

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(private router: Router, private dataService: DataService) {

  }

  canActivate(): boolean {

    if (this.dataService.user.fbid === null) {
      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 fbid of the current user is null, 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 { Routes, RouterModule } from '@angular/router';
import { AuthGuardService } from './services/auth-guard.service';

const routes: Routes = [
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'home', loadChildren:
    './home/home.module#HomePageModule', canActivate:
    [AuthGuardService] },
  { path: 'about', loadChildren:
    './about/about.module#AboutPageModule', canActivate:
    [AuthGuardService] },
  { path: 'login', loadChildren:
    './login/login.module#LoginPageModule' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  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

This 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 PouchDB and Cloudant which we will tackle later, for now, we just want to create the ability for the logged in user to add messages to the screen.

We will also be tying 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:**

```
sendMessage(): void {  
  
  if(this.dataService.user.fbid !== null){  
    //  
  }  
}
```

```

let message = {
    '_id': new Date().toJSON(),
    'fbid': this.dataService.user.fbid,
    'username': this.dataService.user.username,
    'picture': this.dataService.user.picture,
    'message': this.chatMessage
};

this.dataService.addDocument(message);
this.chatMessage = '';

}

}

```

All we are doing here is creating a message object with a few different properties. We add all of the information we just recently grabbed from Facebook, along with the message the user entered. We also add an `_id` field which is for the PouchDB & Cloudant integration we'll be working on in the next lesson, so just ignore that for now.

Once we've created the message we send it off to the `addDocument` method in our data service (which we will implement soon) and reset the chat message input field (so that the input field is ready for the user to type their next message).

NOTE: Your application will trigger an error if you attempt to serve it before implementing the `addDocument` 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 dataService.messages">
    <ion-avatar [attr.slot]="message.username == dataService.username ? 'start' : 'end'">
      <img [src]="message.picture">
    </ion-avatar>
    <div class="chat-message">
      <strong>{{message.username}}</strong>
      <p>{{message.message}}</p>
    </div>
  </ion-item>

</ion-list>
```

Now we're looping through and displaying all of our messages in their own `<ion-item>`.

Remember that the `" syntax is a shortcut for creating embedded templates, and 'ngFor'` will create an embedded template for every **message** in **dataService.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 of course.

We mostly have a pretty basic template set up, but we are doing something a little funky:

```
<ion-avatar [attr.slot]="message.username == dataService.username  
? '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 data 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" class="logo" />
  </ion-title>
</ion-toolbar>
</ion-header>

<ion-content padding>
  <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-content>

```

This is a really simple page and there's not really anything exciting here. We've implemented the `<ion-menu-button>` button just as we did on the home page, and we've just added a bit of descriptive content.

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. Keep in mind that the application will not actually compile properly now as we are referencing a method in the `dataService` 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 PouchDB and Cloudant, which we will be doing in the next lesson, and then we just need to tidy things up and make it look pretty.

Lesson 6: Local and Remote Backend with PouchDB and Cloudant

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 users messages both locally and to a remote backend.

PouchDB is an in browser NoSQL database that was inspired by the CouchDB project. It's biggest feature is that it allows for data storage offline, and it can automatically sync to a remote database when the application comes back online. Like using the SqlStorage service that Ionic provides, using PouchDB will ensure that your local data won't be randomly wiped out like it can be when using plain local storage.

Teaching NoSQL is far beyond the aims of this book, but to give you a really quick background, NoSQL databases usually store data in a JSON like key value style format, instead of the relational style tables that traditional SQL uses. The two approaches are completely different, the way you use them is different and the way you need to think about them is different. So, when it comes to using NoSQL you have to drop a lot of the preconceived notions you would have from SQL (assuming you've used that in the past). The data we're storing is super simple though, so we don't need to worry about learning how to use a NoSQL database properly.

So PouchDB will handle storing our data locally, but we are also going to be using Cloudant to create a remote database. We will be syncing the local PouchDB database to the remote Cloudant database, so that whenever the application is online it can fetch new data from Cloudant, and any new local data will also be pushed to Cloudant. Fortunately,

these two tools are designed to work with each other and setting up remote syncing, which is usually a very complex task, is super simple.

Cloudant is a DBaaS (Database as a Service) so we are going to have to create an account (which is free for low usage) to use it, as well as set up the database. Using something like Cloudant is really easy and scales very well (since all the backend architecture is handled for you), but if you prefer you can easily use something like CouchDB or Couchbase installed on your own server to sync with PouchDB. I won't be covering how to set those up, but it's not all that different.

Creating a Cloudant Database

Before we start jumping into the code, we're going to set up our backend on Cloudant through IBM Bluemix. Bluemix gives you access to IBM's Open Cloud Architecture and can be used to create, deploy and manage cloud based applications. It provides a bunch of services you can use and one of those is Cloudant.

First you will need to [create an IBM Cloud account](#) or log into your existing account. Once you have created your account and logged in to your dashboard, you will be able to click on **Cloud Foundry Services** and then you should see a screen with a **Create Resource** button. Click that and choose the **Cloudant** option (there are quite a few options here, so you might want to do a text search for 'Cloudant').

You will then need to fill out the details for the resource, which should look something like this:

The screenshot shows the IBM Cloudant service creation interface. At the top, there's a summary bar with the following details:

- Service name:** Cloudant-my-test
- Choose a region/location to deploy in:** Sydney
- Choose an organization:** joshua.morony@gmail.com
- Choose a space:** dev

Below this, there are sections for **AUTHOR** (IBM), **PUBLISHED** (07/09/2018), and **TYPE** (Service). To the right, it says "Monthly prices shown are for country or region: Australia".

The main area is titled "Pricing Plans" and contains a table:

PLAN	FEATURES	PRICING
Lite	1 GB of data storage Provisioned throughput capacity fixed at: 20 Lookups/sec 10 Writes/sec 5 Queries/sec	Free
Standard	20 GB of free data storage Throughput capacity scales in fixed ratio of: 100 Lookups/sec 50 Writes/sec 5 Queries/sec	A\$1.134 AUD/GB of data storage A\$0.2835 AUD/Lookup per second A\$0.567 AUD/Write per second A\$5.67 AUD/Query per second

Notes below the table state: "The Lite plan provides access to the full functionality of Cloudant for development and evaluation. The plan has a set amount of provisioned throughput capacity as shown and includes a max of 1GB of encrypted data storage." and "Lite plan services are deleted after 30 days of inactivity."

You may change some of these options if you wish, but for now you probably just want to use the free 'Lite' tier to play around in. When you are ready, click **Create**. On the following page, click the **Launch Cloudant Dashboard** button to access your new Cloudant database.

On the menu on the left, click the database icon (the stacked discs) and then click the **Create Database** option in the top right to create a new database and call it **camperchat** or whatever you prefer.

The screenshot shows the Cloudant Databases dashboard. On the left, there's a sidebar with icons for databases, users, and settings. The main area is titled "Databases" and shows a table for "Your Databases". The table has columns for Name, Size, # of Docs, and Actions. In the Actions column, there's a "Create Database" button. A modal window is open over the table, titled "Create Database". It has a text input field containing "camperchat" and a blue "Create" button.

Once you have done this, you should be inside of your new database! If you click the **JSON** button in the top right, you will be taken to the URL for your database. It will look something like this:

<https://12345678-4850-4d82-bcac-92f41b686e65->

`bluemix.cloudant.com/camperchat/_all_docs`

Cloudant/CouchDB databases can be interacted with directly through HTTP requests, and this URL can be used to access all of the documents currently in the database. By default, these databases are publicly accessible, but it is possible to set up private databases with authentication requirements as well (as I mentioned, this is too complex for the scope of this book).

You should also go to the **Permissions** of your dashboard and generate an API key – this can be used with PouchDB to access your database, and it's a better idea to use an API key which is revokable rather than the actual username and password for your account. Just click **Generate API Key** and make sure to take note of the password because once you leave the screen you won't be able to find it again. You should also make sure to tick the `_writer` and `_replicator` options for the API key you just generated - these permissions will be required in order to add messages to the database, and to replicate/sync the remote database to the local PouchDB database in our application.

IMPORTANT: If you include an API key directly in your code for the application, it is possible for people to find that key by looking through your source code. This may or may not be a problem depending on the type of application you are creating, but just keep in mind that your source code is not secure. If you need access to the database to be restricted, you may want to investigate a different method like creating different API keys for each user, or only providing access to the database through a server that sits in between the front end and the database (since you can store credentials secretly on the server).

There's just one more thing we need to do in here. We need to enable CORS (Cross Origin

Resource Sharing) so that we are able to make requests to the database from our application. To do that go to **Account** in the left menu, select **CORS** and then choose the All Domains (*) options.

You should now have everything you need set up on Cloudant, ready to be used with PouchDB.

Integrating PouchDB

Earlier on we created a Data service which so far we are only using to store a few values from the Facebook API. Now we're going to extend that Data service to handle storing and retrieving data with PouchDB.

> **Modify src/app/services/data.service.ts to reflect the following:**

```
import { Injectable } from '@angular/core';
import * as PouchDB from 'pouchdb/dist/pouchdb';

interface Message {
  _id?: string;
  username: string;
  message: string;
  picture: string;
}

interface UserData {
  fbid: number;
```

```
username: string;
picture: string;
}

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

  public user: UserData = {
    fbid: null,
    username: null,
    picture: null
  }

  public messages: Message[] = [];

  private db: any;
  private cloudantUsername: string;
  private cloudantPassword: string;
  private remote: string;

  constructor(){

    // IMPORTANT: This is just a simple implementation example
    // for a publicly accessible database. The username and password
    // used here should be an API key
    // provided by cloudant (not your account username and
  }
}
```

password). Any key that is stored inside of your Javascript code is accessible to everybody, so you should

```
// never put any sensitive information your client side code.  
// Implementing a database that is secure/not publicly  
accessible will require you to use your own proxy server to hide  
these details (this is much more advanced).
```

```
this.db = new PouchDB('camperchat');  
this.cloudantUsername = 'YOUR API KEY USERNAME'; // NOT your  
account username  
this.cloudantPassword = 'YOUR API KEY PASSWORD'; // NOT your  
account password  
this.remote = 'https://YOUR-CLOUDANT-  
URL.cloudant.com/camperchat';  
  
//Set up PouchDB  
let options = {  
    live: true,  
    retry: true,  
    continuous: true,  
    auth: {  
        username: this.cloudantUsername,  
        password: this.cloudantPassword  
    }  
};  
  
this.db.sync(this.remote, options);
```

```
}

addDocument(message): void {

}

getDocuments(): Promise<any> {
    return Promise.resolve(true);
}

handleChange(change): void {

}

}
```

We've already installed PouchDB with **npm** so to make it available in this service we have just added the following line:

```
import * as PouchDB from 'pouchdb/dist/pouchdb';
```

In the constructor we are just handling the set up of PouchDB and the sync to the remote Cloudant database. First we create a new PouchDB, or get a reference to an already existing one:

```
this.db = new PouchDB('camperchat');
```

and then we define a few variables which will be used to connect to the Cloudant database:

```
this.cloudantUsername = 'YOUR API KEY USERNAME'; // NOT your account username  
this.cloudantPassword = 'YOUR API KEY PASSWORD'; // NOT your account password  
this.remote = 'https://YOUR-CLOUDANT-URL.cloudant.com/camperchat';
```

Make sure to replace these values with your own from the Cloudant dashboard (make sure that you don't include _all_docs on the end of your remote address). Next we create an options object to configure our connection to the Cloudant database and then we call the sync method:

```
this.db.sync(this.remote, options);
```

This will set up replication from our PouchDB database to the Cloudant database, and it will also set up replication from the Cloudant database to the PouchDB database. So now if we add some data to our PouchDB database it will automatically be reflected in the

remote Cloudant database, and if we change or add some data in the remote Cloudant database it will automatically be reflected in our local database.

After that we have just created three empty functions, which we are going to go through implementing now.

addDocument

> Modify the addDocument function to reflect the following:

```
addDocument(message): void {  
    this.db.put(message);  
}
```

To add a document (a data object in NoSQL terms is called a "document", so think of a document as a bit of data, not something like a Word document) to our PouchDB database we simply call put on our database. So we will be able to pass this function any object and it will add it to the database.

getDocuments

> Modify the getDocuments function to reflect the following:

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

```
return new Promise(resolve => {

  this.db.allDocs({
    include_docs: true,
    limit: 30,
    descending: true
  }).then((result) => {

    this.messages = [];

    let docs = result.rows.map((row) => {
      this.messages.push(row.doc);
    });

    this.messages.reverse();

    resolve(this.messages);

    this.db.changes({live: true, since: 'now', include_docs: true}).on('change', (change) => {
      this.handleChange(change);
    });
  }).catch((error) => {
```

```
    console.log(error);

  });

}

}
```

This function handles retrieving all of the documents from our database (remember, a document is just what a data object is called). This is an asynchronous operation, so we wrap it in a promise which resolves when the data is returned. This will allow us to use the `getDocuments().then()` syntax elsewhere in the application. By calling `allDocs` every document will be retrieved, but we are supplying a few options here:

```
include_docs: true,
limit: 30,
descending: true
```

The `include_docs` option here may seem a little confusing, but we need to specify this so that all the data in our documents are returned (message, picture etc.). If this option is left out then only the id of the document is returned. We also set a limit of 30 and a descending order so that only the 30 latest documents (chat messages) are returned.

We pass the result of that operation through to our handler, and for each row that is

returned we push it into a `this.data` array. We want these results to be in reverse order though because we want to show the latest message at the bottom so we flip the array with `reverse`. After this we `resolve` the promise we created and pass back the data, and then we set up a changes listener.

The changes listener will call the `this.handleChange` function every time a change is detected in the database (when another user has added a chat message for example), and the change itself will be passed into that function. We'll define that now.

handleChange

> **Modify the `handleChange` function to reflect the following:**

```
handleChange(change): void {  
  
  let changedDoc = null;  
  let changedIndex = null;  
  
  this.messages.forEach((doc, index) => {  
  
    if(doc._id === change.id){  
      changedDoc = doc;  
      changedIndex = index;  
    }  
  
  });  
}
```

```

//A document was deleted
if(change.deleted){
    this.messages.splice(changedIndex, 1);
}
else {

    //A document was updated
    if(changedDoc){
        this.messages[changedIndex] = change.doc;
    }
}

//A document was added
else {
    this.messages.push(change.doc);
}

}

```

What we want to do with the change that is passed in is reflect it in our `this.data` array, but it gets a little bit tricky. The change object that is sent back could either be a document that has been updated, a new document, or a deleted document.

Detecting a deleted document is easy enough because it will contain the **deleted** property.

But to see if it is an update, we need to check if we already have a document with the same id (and update that one if we find it), if we don't then we know it is a new document (and need to add it to the array).

We have our Data service completely set up now, but there's still a couple more things we need to do to make use of it. We will need to trigger the initial load of the documents from the database, and we also want our chat list to automatically scroll down to the bottom of the list when new documents are added.

> **Modify src/app/home/home.page.ts to reflect the following:**

```
import { Component, ViewChild, ElementRef, OnInit } from
'@angular/core';
import { IonContent, IonList } from '@ionic/angular';
import { DataService } from '../services/data.service';

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

  @ViewChild(IonContent) contentArea: IonContent;
  @ViewChild(IonList, {read: ElementRef}) chatList: ElementRef;

  public chatMessage: string = '';
}
```

```
private mutationObserver: MutationObserver;

constructor(public dataService: DataService){

}

ngOnInit(){

    this.mutationObserver = new MutationObserver((mutations) => {
        this.contentArea.scrollToBottom(200);
    });

    this.mutationObserver.observe(this.chatList.nativeElement, {
        childList: true
    });

    this.dataService.getDocuments().then((data) => {
        console.log("chat loaded");
    }, (err) => {
        console.log(err);
    });
}

sendMessage(): void {

    if(this.dataService.user.fbid !== null){

```

```

let message = {
  '_id': new Date().toJSON(),
  'fbid': this.dataService.user.fbid,
  'username': this.dataService.user.username,
  'picture': this.dataService.user.picture,
  'message': this.chatMessage
};

this.dataService.addDocument(message);
this.chatMessage = '';

}

}

```

In the ngOnInit method we are now calling the getDocuments() function to load in the messages data. The more interesting part is the addition of a "mutation observer" - remember when I said we were going to make use of those references we go 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> 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. 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).

That's it! You should now have a fully functional chat application. It works, but it's still pretty ugly. In the next lesson we are going to make it look prettier and even set up some animations!

Lesson 7: 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: #5b91da
```

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-menu {  
  
    ion-content {  
        --background: url('/assets/images/login-  
background.png') 40%;  
        text-align: center;  
    }  
  
    ion-list {  
        --ion-item-background: transparent !important;  
        margin-top: 44px;  
    }  
  
    ion-item {  
        color: #fff;  
    }  
}
```

We're modifying the position of the logo a bit, setting a background image that will cover the menu, 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.scss to reflect the following:**

```
ion-label {  
  white-space: normal;  
}  
  
ion-row {  
  padding: 0;  
  margin: 0;  
}  
  
ion-col {  
  padding: 0;  
  margin: 0;  
}  
  
ion-footer ion-button {  
  float: right;  
}  
  
ion-list {  
  background-color: #f3f3f3;  
}
```

```
textarea {
    width: 98%;
    margin: 0px 10px;
    padding: 5px;
    background-color: #fff;
    font-size: 1.2em;
    resize: none;
    border: none;
}

.chat-message {
    background-color: #fff;
    width: 100%;
    padding: 10px;
    border-radius: 10px;
    margin: 10px 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:**

```
ion-content {
```

```
--background: url('/assets/images/login-background.png') 40%;  
text-align: center;  
}  
  
.login-description {  
height: 60%;  
line-height: 2em;  
}  
  
.login-description p {  
color: #fff;  
}  
  
.login-button {  
box-shadow: 0px 6px 20px 0px rgba(0,0,0,0.18);  
}
```

We are again setting a background image (the same one used in the menu) but this time it will be applied to the entire login page. We position the description content and also add a bit of a shadow to the Facebook login button.

That's it for the basic styling, the pages in your application should now look something like this:



CamperCHAT

aaa

test

aaa

aaa

Joshua Morony
Hello

Joshua Morony
Is it me

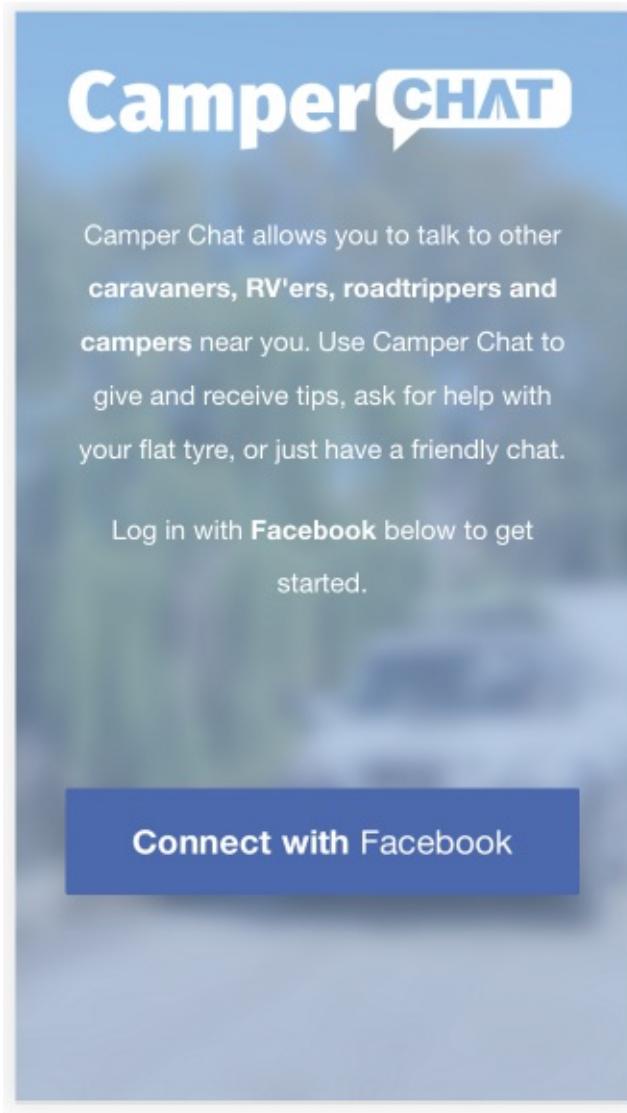
Joshua Morony
Youre looking for

Joshua Morony
Aaaaaaa

Joshua Morony
Halllooo

enter 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 users 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 in to 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 dataService.messages">  
        <ion-avatar [attr.slot]="message.username ==  
dataService.username ? 'start' : 'end'" class="animate-in-  
primary">
```

```
<img [src]="message.picture">
</ion-avatar>
<div class="chat-message animate-in-secondary">
  <strong>{{message.username}}</strong>
  <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 tutorial. We've learned a lot through developing this application, but the main take aways are:

- Navigation
- Using a Sliding Menu
- Using PouchDB to store local data
- Using Cloudant to store remote data
- Using the Facebook API for authentication and other features
- Updating and displaying data in real time

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's a few ideas to try out:

- Make the users device vibrate when a message is received by using a Cordova plugin **[EASY]**
- Add "Channel Categories" that will allow the user to switch between different chat rooms, e.g: Vehicle Issues, General Chat, Camping **[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.

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 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 any Cordova plugins, they won't 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:

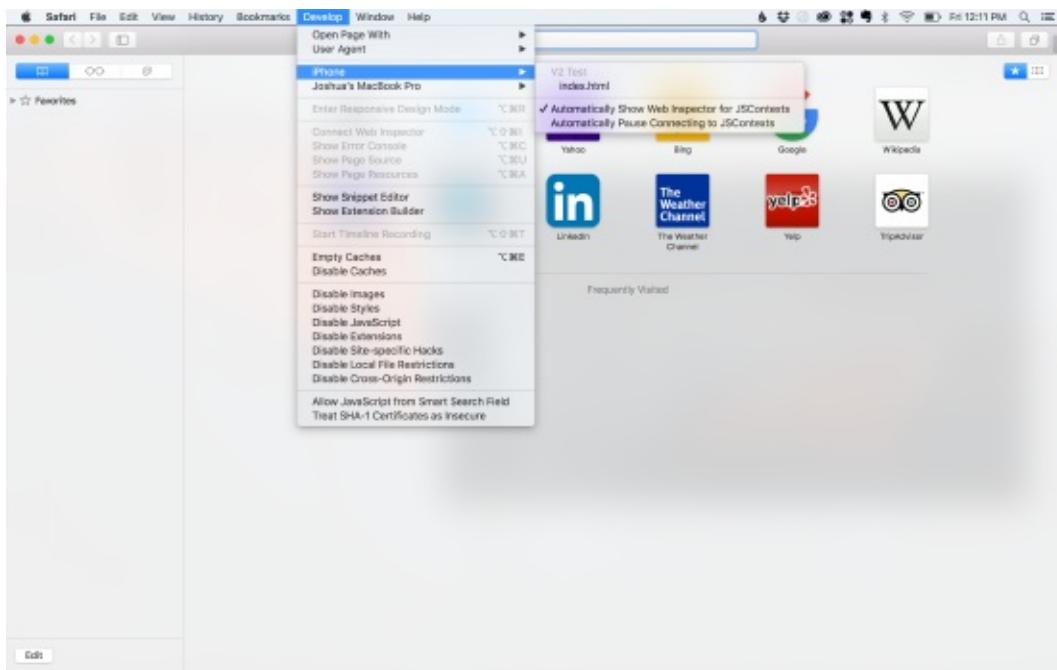
- Go to Settings > Safari > Advanced and turn Web Inspector and Javascript on

You will also need to enable the Develop menu in Safari (on your computer) by going to:

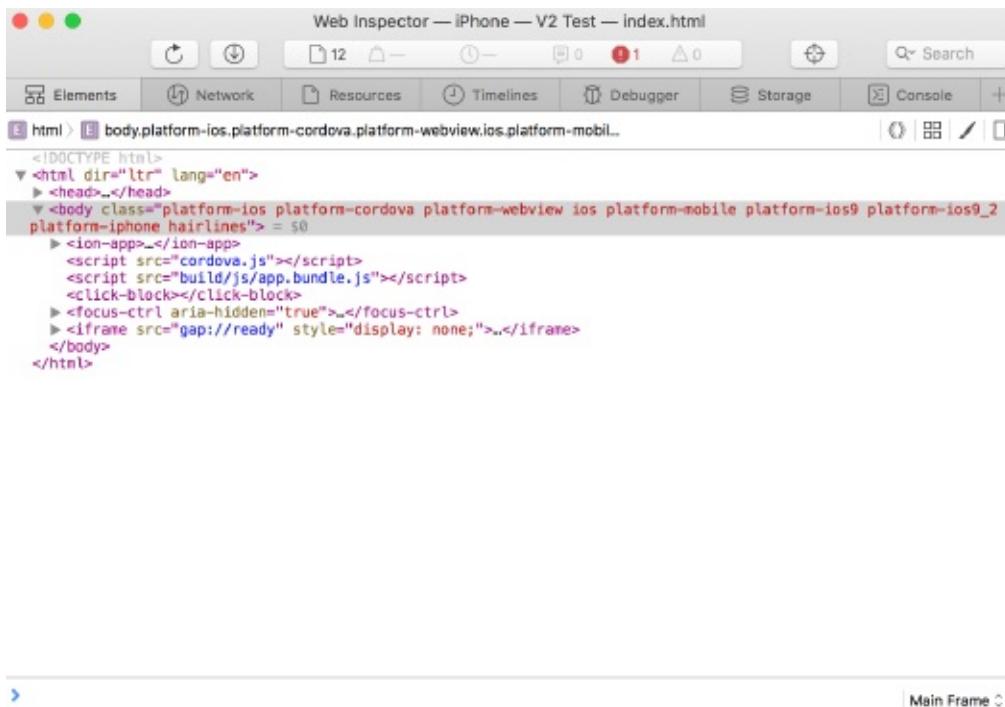
Safari > Preferences > Advanced > Show Develop menu in menu bar

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.

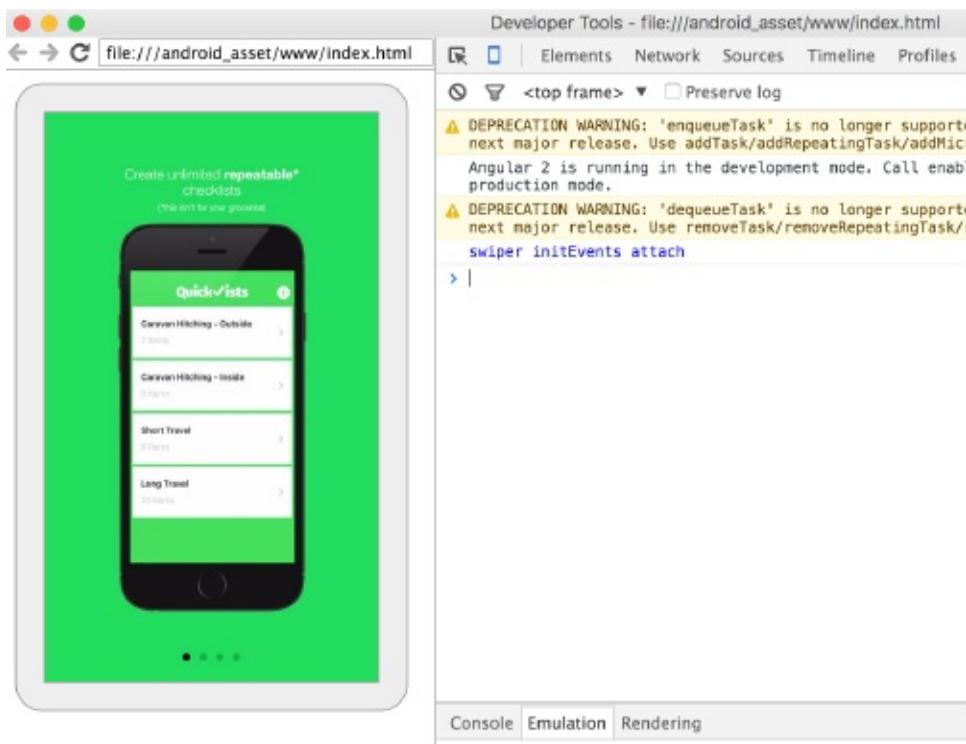
You will need to have USB debugging enabled on your device for this to work. Enabling USB debugging is different for different devices, so just Google "[YOUR DEVICE] enable usb debugging". On your desktop browser (in Chrome) you will now be able to go to the

following address:

chrome://inspect/#devices

The screenshot shows the Chrome DevTools interface with the 'Devices' tab selected. On the left, a sidebar lists 'Devices', 'Pages', 'Extensions', 'Apps', 'Shared workers', 'Service workers', and 'Other'. The main area displays a device entry for 'Blade S6 #77402CB5'. Below the device name, it shows the application 'io.ionic.starter'. Underneath the application name, there is an unchecked checkbox labeled 'Ionic file:///android_asset/www/index.html inspect'. The browser's address bar at the top shows '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.

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 copy over the new build to your Capacitor project:

```
ionic cap copy
```

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 ends 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.

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. There is a command built directly into the Ionic CLI to help generate resources, but at the moment this requires Cordova integration with your project. I suspect this will change in the future but it's a little awkward to use right now. 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
- [Ionic Resources Command](#)

- [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

These are just a few of the options available. For this example, I used **MakeAppIcon** for the iOS icons, **ImageGorilla** for the Android splash screens, I generated the iOS splash screens manually (as I was just using a universal size), and I had Android Studio create the Android icons for me automatically.

Now, let's move on to adding the assets to a Capacitor project. At this point, I will assume that you already have a Capacitor project created and have added your desired platforms, e.g:

```
ionic cap add ios
```

```
ionic cap add android
```

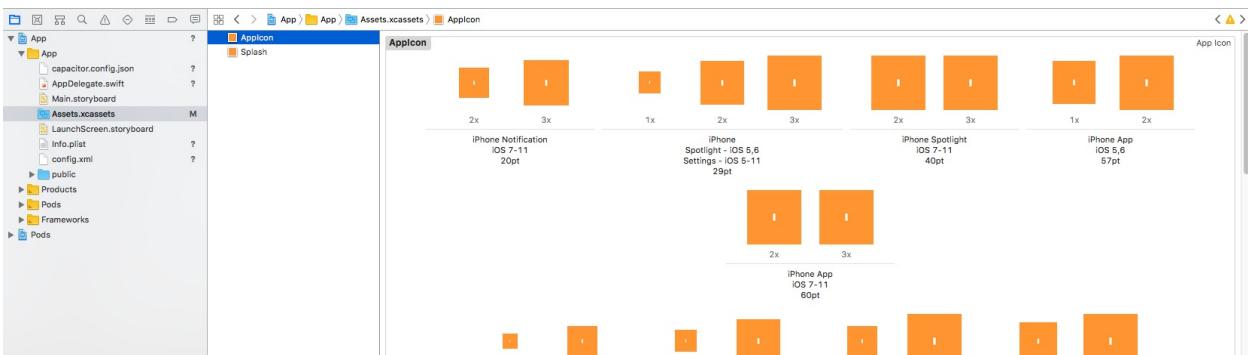
If you haven't, you should add your desired native platforms now.

iOS Icons

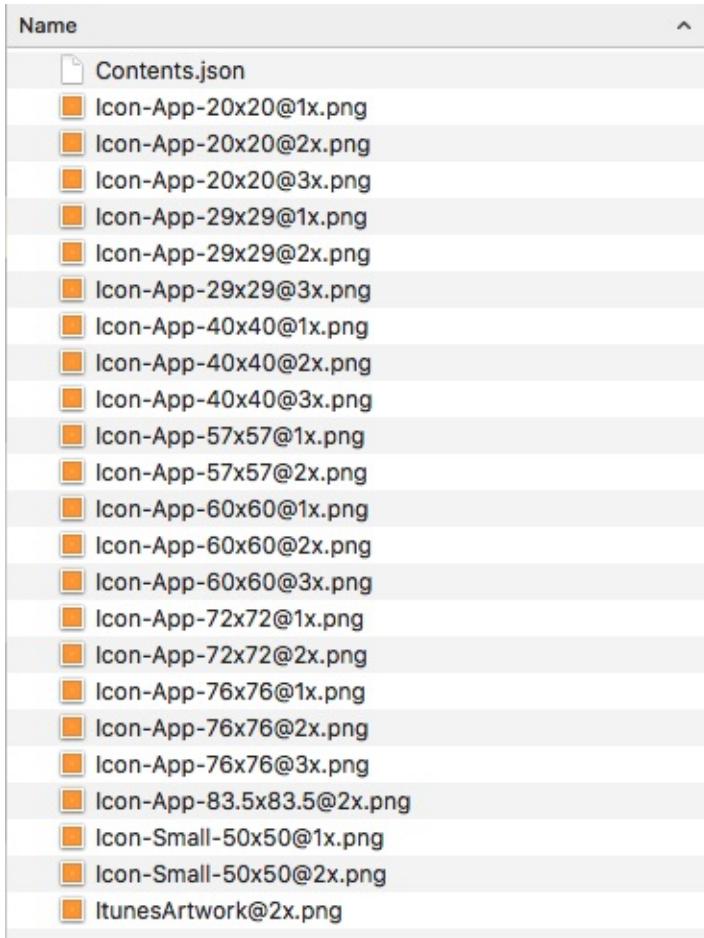
There are a couple of ways we can approach adding iOS icons to a project. First, let's find where the existing icons live. Open up your Capacitor project in Xcode:

```
ionic cap open ios
```

Navigate to **App > App > Assets.xcassets** and you will be able to see the splash and icon assets for your project. As you can see from the image below, I have replaced all of the icon assets with my own icon:



You can either do this directly through the Xcode interface by dragging your icons into the appropriate slot, or you can overwrite them directly through the file system. You can right-click on **AppIcon** and choose **Show in Finder** to reveal where these icons live on the file system. If you open **AppIcon.appiconset** you will see all of the icons:



If you used a tool that generates the `Contents.json` file for you as well as the icons, then you can just replace everything in this folder with your new icon set. If you don't have a `Contents.json` file you would either need to replace each icon making sure to keep the same name and size, or update the existing `Contents.json` to reflect the new file. If you are going this route then it may just be easier to drag and drop into the Xcode interface so you don't need to worry about messing with the `Contents.json` file.

iOS Launch Images

Supplying the launch images is basically the same as the icons. You would have also seen a **Splash** entry under **Assets.xcassets**, and you can also reveal these files in your file

system by right-clicking on **Splash**. You should take the same approach here - either update the images in the file system or drag your new splash screen into the Xcode interface. Since there are only three images here, it's actually also quite easy to just open up the .png files through the file system, edit them, and resave them with your desired splash screen.



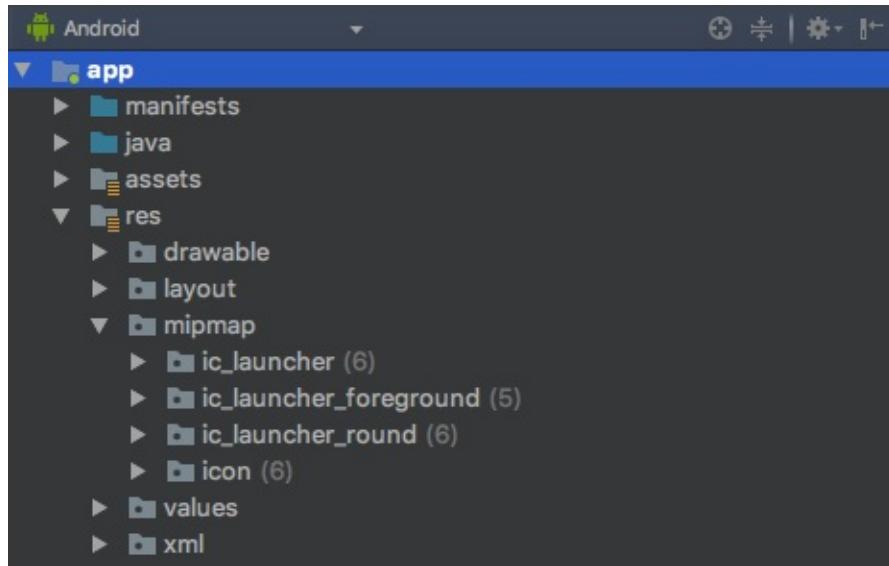
The benefit of using a universal image for the launch image is that you only need to create a single design, and that can be used across all devices. The downside to this is that the image will be cropped to fit devices, so it's important that the important parts of your design are centered. If you would prefer, you can create specific designs for each of the available resolutions to make better use of the available space.

Android Icons

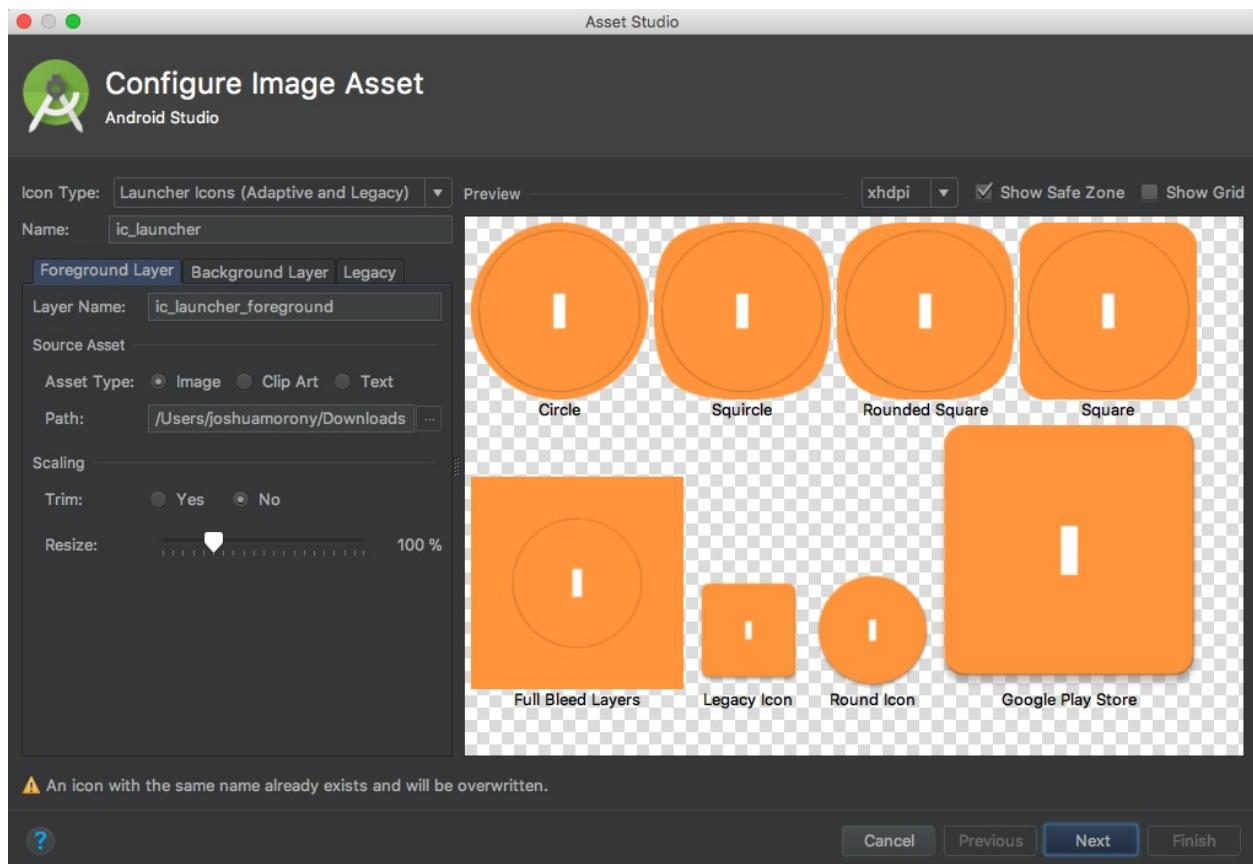
Generating icons for an Android project is actually quite easy. First, you will need to open your Capacitor project in Android Studio:

```
ionic cap open android
```

The icons that will be used in your project will live inside of the **mipmap** folder, which you can find at **app > res > mipmap**:



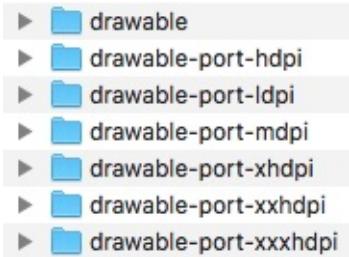
To generate a new set of icons, you should right-click on the **res** folder and go to **New > Image Asset**:



All you need to do is select the **Image** option and then select your icon (which should be at least 1024x1024 pixels). Once you have chosen your icon, click **Next** and then **Finish**. I have just used an ugly icon as an example, but if you are interested in learning more about how Google recommends you design your icons, you should [read this](#).

Android Splash Screen

The last thing we need to do is update the splash screens for Android. You can find the default splash screens that Capacitor sets up at **app > res > drawable > splash**. If you right-click on the **drawable** folder and choose **Reveal in Finder** you will be able to see folders for all of the various resolutions available:



You will need to replace the `splash.png` file in each of these `drawable` folders with your own. If you have used a generator that automatically generates this file structure, you may be able to just drag and drop them all in at once rather than doing it one by one.

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.

I suspect that this will all be neatly integrated into the Ionic or Capacitor CLI in the future, but for now, there is a bit of manual work involved. 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 [Ionic Package](#) to handle creating the iOS build for you. **NOTE:** This is not currently available for Capacitor, but it is planned.
- Using a service like [macincloud](#) to access a Mac machine remotely

- Borrow a friend's 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. There is a lesson later in the book that covers how to create the necessary certificates and provisioning profiles for iOS from a Windows machine - so 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.

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. In this example, we are using version 9.3 of Xcode.

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](#) library. 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 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:

```
npm install -g n
```

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

```
n 8.11.1
```

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.

5. Install Capacitor

You should also have Capacitor installed at this stage, and you should have added the native platforms that you are targetting. 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 you supply to Capacitor during the configuration step (e.g. `com.example.yourapp`). You will need to use this later to create an identifier for your application with Apple.

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
```

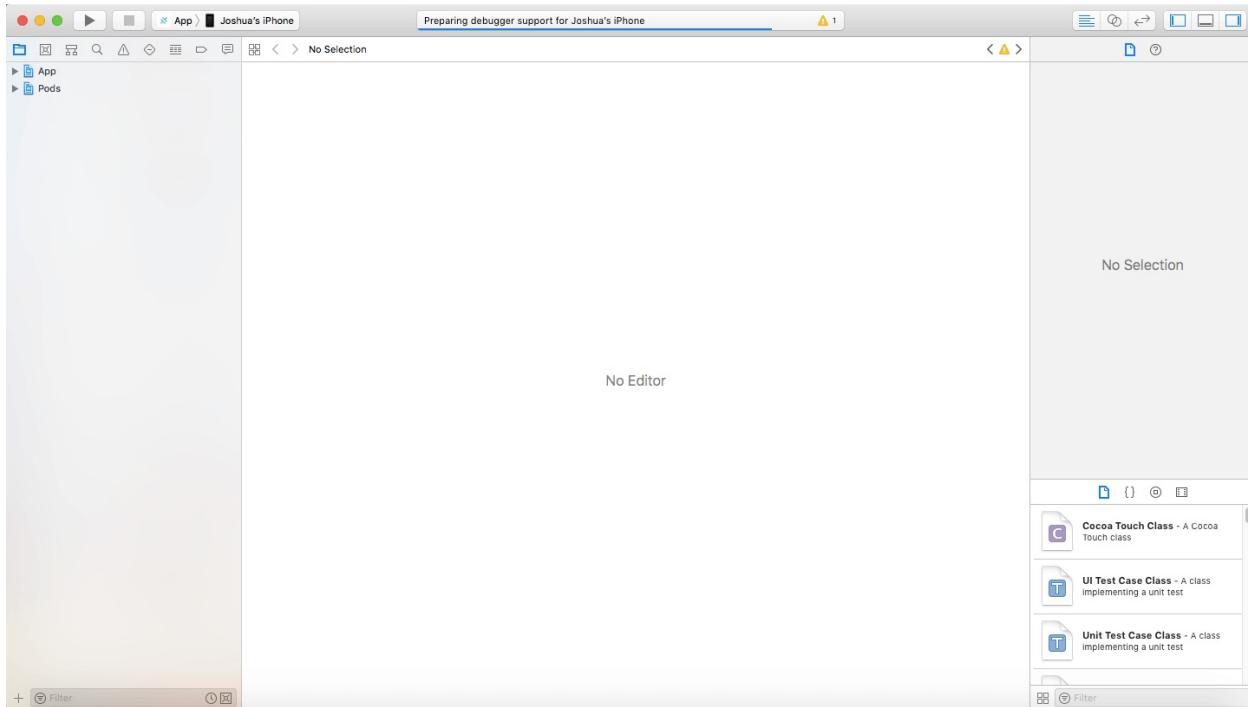
You do not need to run the sync command each time you make changes to your project.

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
```

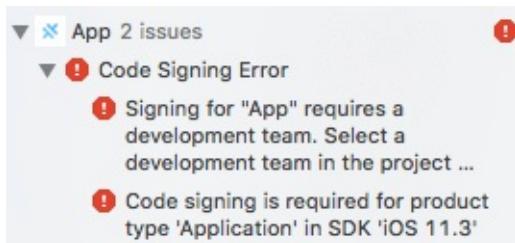


From this point onwards, we are basically just dealing with a normal native project. We just have our web code bundled in there.

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:

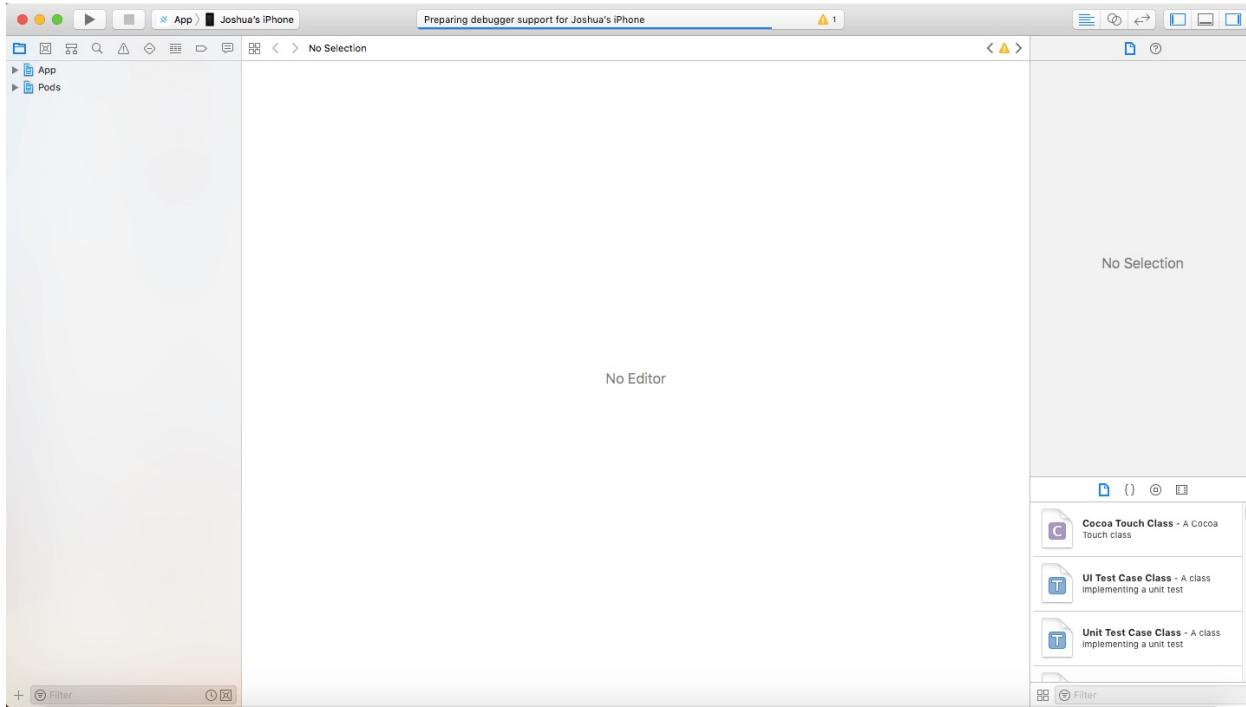
`Signing for "App" requires a development team. Select a development team in the project...`

`Code signing is required for product type 'Application'`

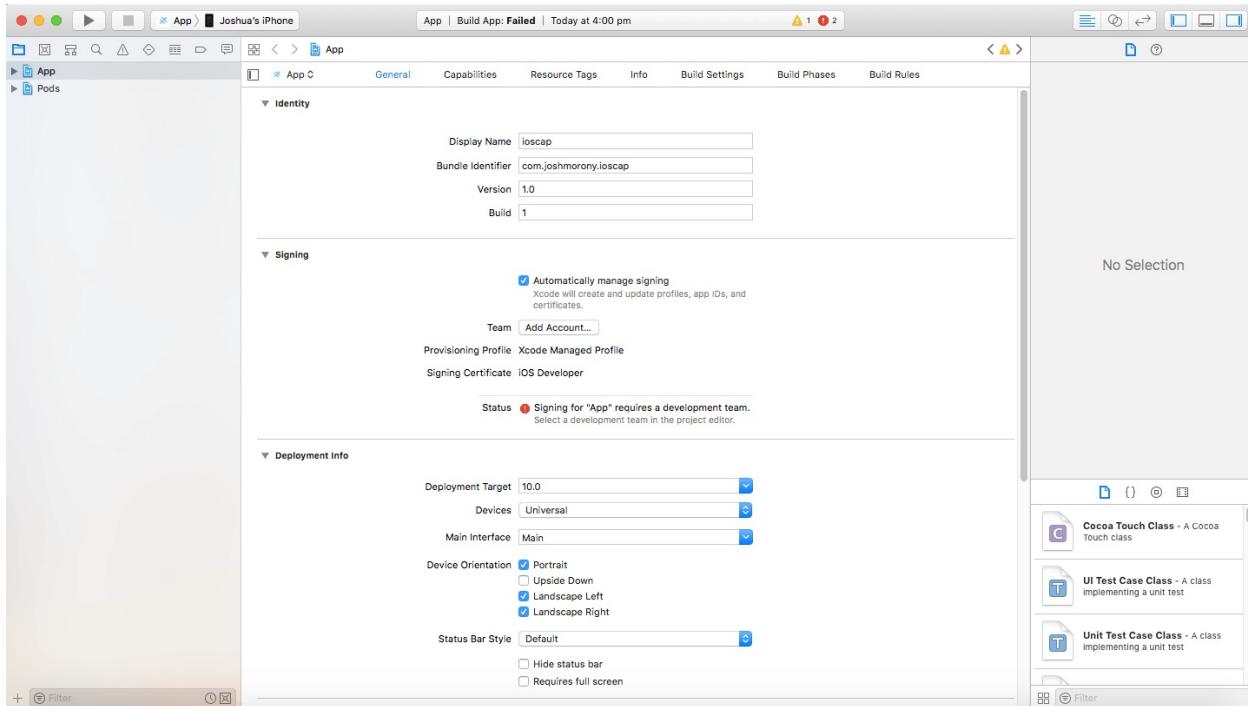
Apple requires that applications are "signed" in order to run on a device (and to be distributed on the App Store). This is basically a way to prove ownership of that application and to associate it with a particular Apple ID. In order to fix this error, you will need to set up signing for the application.

7. Sign the Application

To set up signing for the application, click **App** in the file explorer for the project in Xcode:

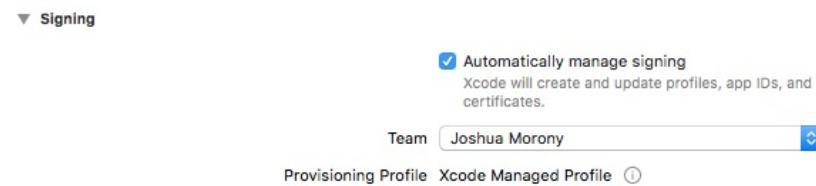


This will bring you to a page that looks like this:



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:



Hit the 'Play' button again, and this time the application should be installed on your device.

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` 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. Under **Identifiers**, go to the **App IDs** section and hit the '+' button:

The App ID string contains two parts separated by a period (.) — an App ID Prefix that is defined as your Team ID by default and an App ID Suffix that is defined as a Bundle ID search string. Each part of an App ID has different and important uses for your app. [Learn More](#)

Name: Capacitor Example
 You cannot use special characters such as @, &, *, ''

Value: 9YABKUX5J7 (Team ID)

App ID Suffix

Explicit App ID
 If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID: com.joshmorony.example
 We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

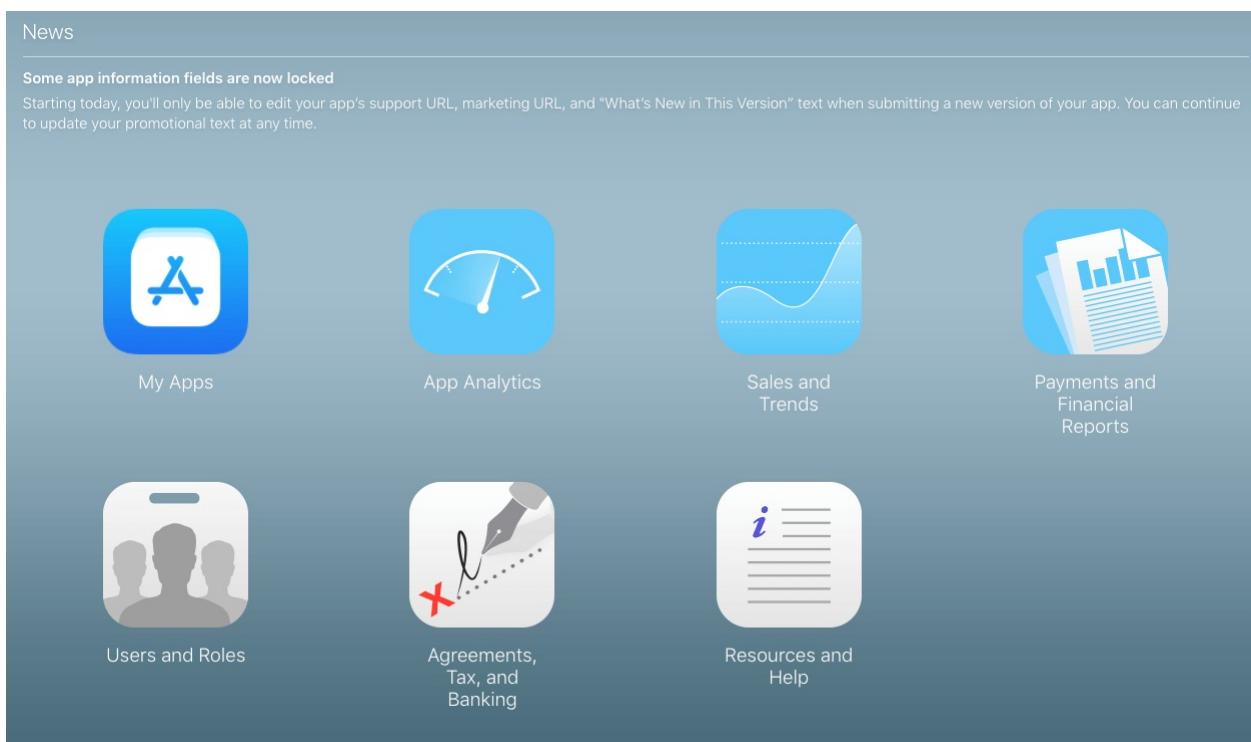
Fill out the details on this screen. When entering an explicit App ID, make sure that it matches the ID that you supplied to Capacitor in the project initialisation (e.g. com.example.yourapp). Make sure that you also enable any services that your app requires. When you are ready, click **Continue**.

Review the information on the next screen, and hit **Register**.

2. Create an Application in iTunes Connect

In order to submit an application, we need to create a record for it in [iTunes 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 [iTunes 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. Fill out the information in the box that pops up:

New App

Platforms ?

iOS tvOS

Name ?

Capacitor Example

Primary Language ?

English (Australia) ▾

Bundle ID ?

Capacitor Example - com.joshmorony.capacitorex ▾

SKU ?

20180430

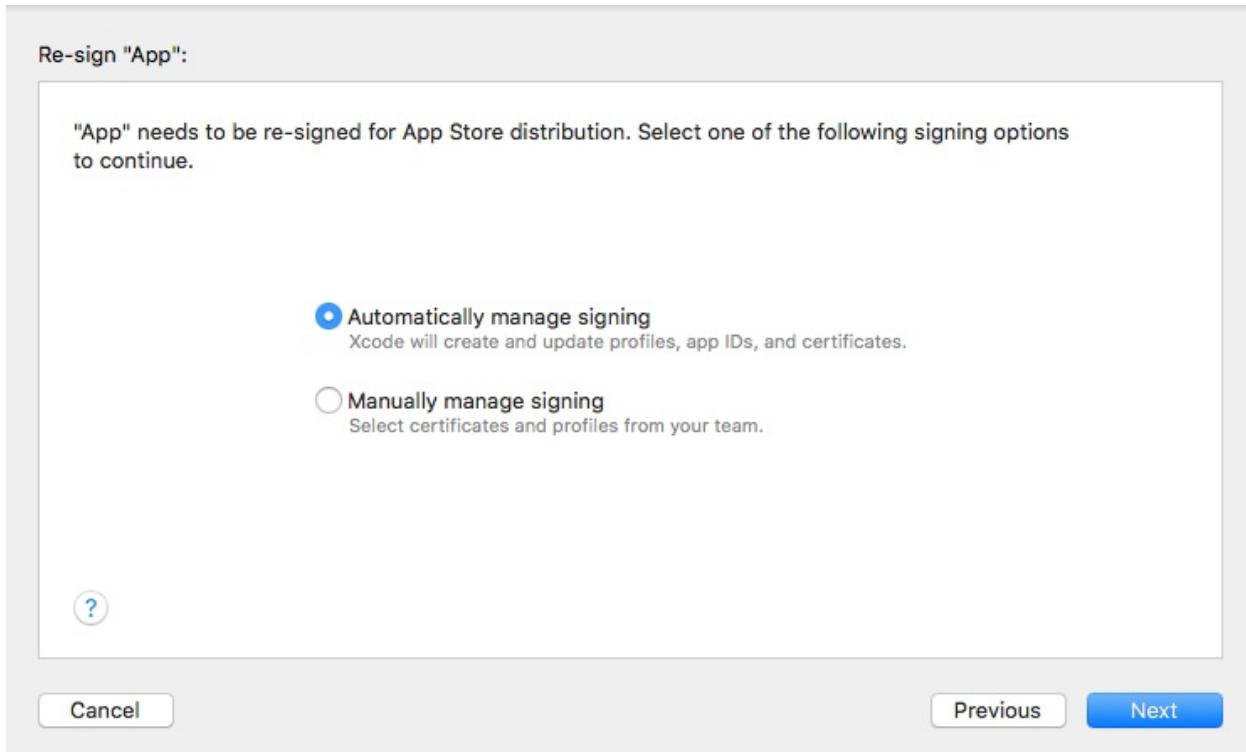
Cancel Create

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 in 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.

3. Sign and Upload the Application to iTunes Connect

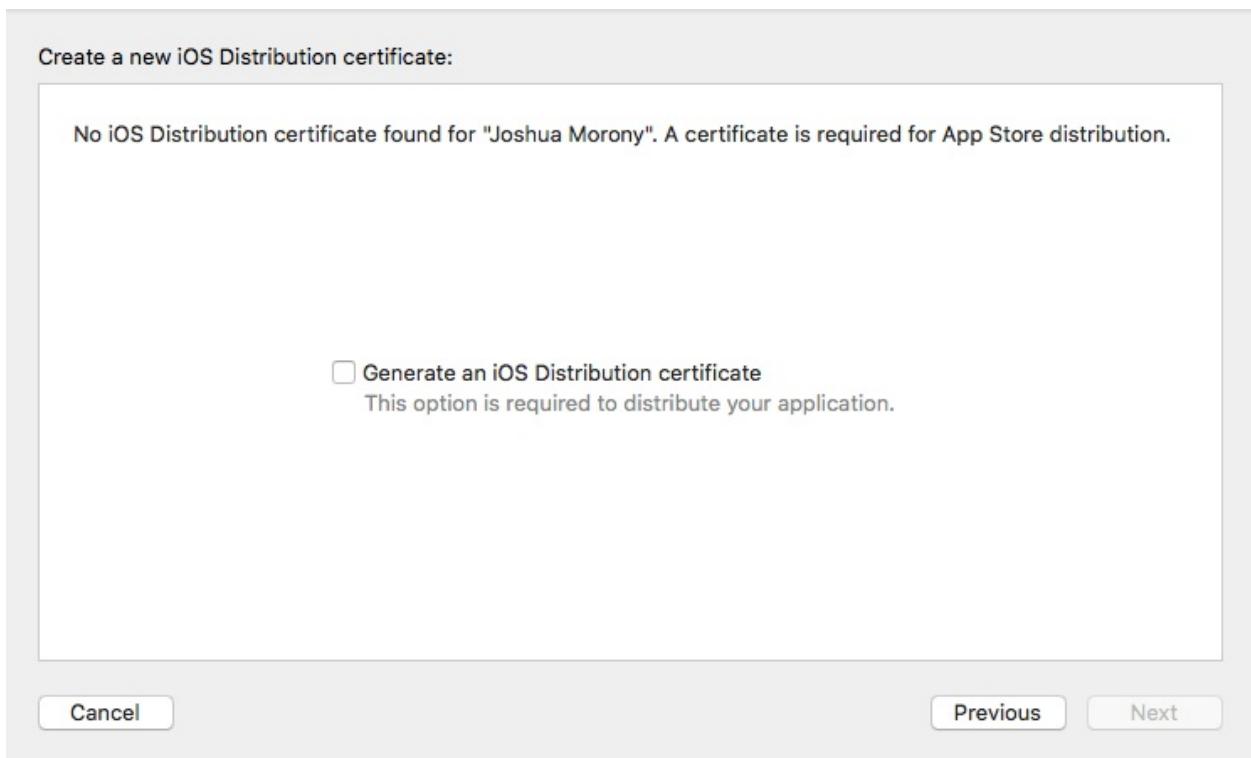
Let's head back to Xcode now. You should change the target device near the 'Play' button back to **Generic iOS Device**. Once you have done this, go to Product > Archive, and you should see a screen with your application should pop up.

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 iTunes Connect (and you should definitely do this to make sure everything is OK before submitting for review). Click **Upload to App Store**.



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.

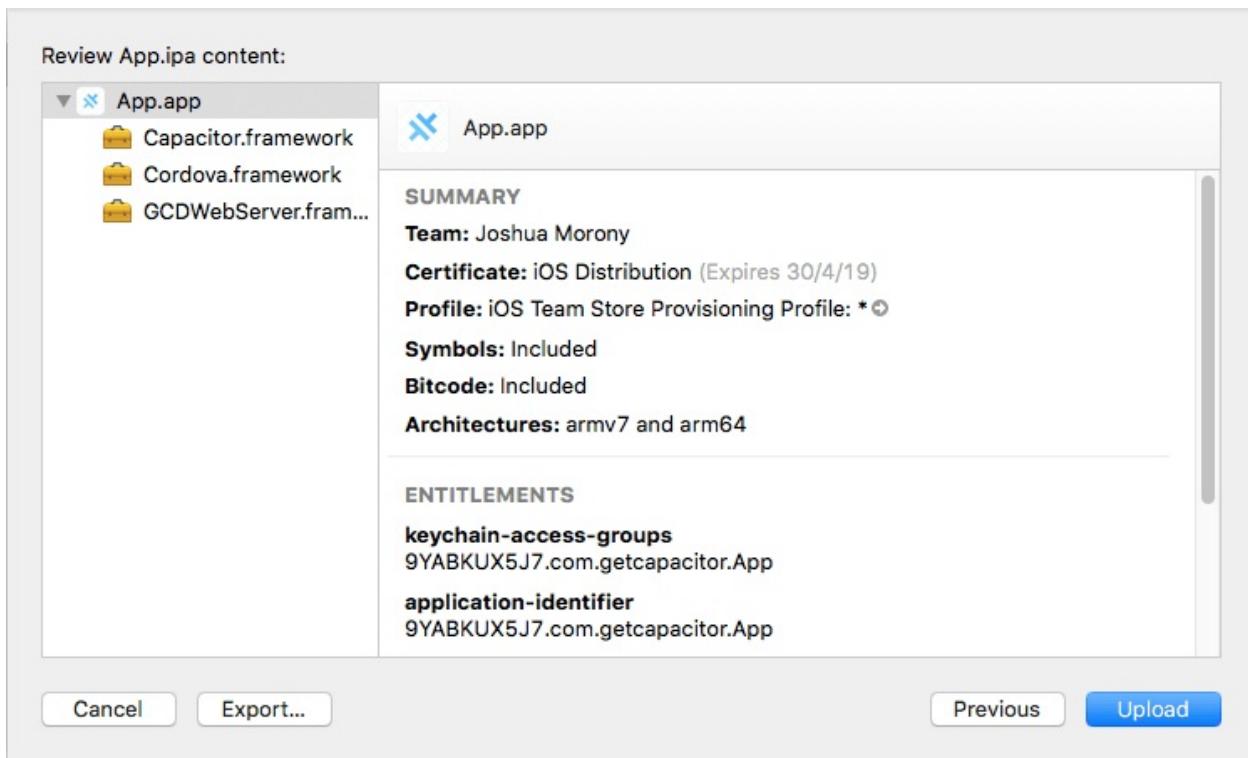
If you do not already have a distribution certificate created, you will see something like this:



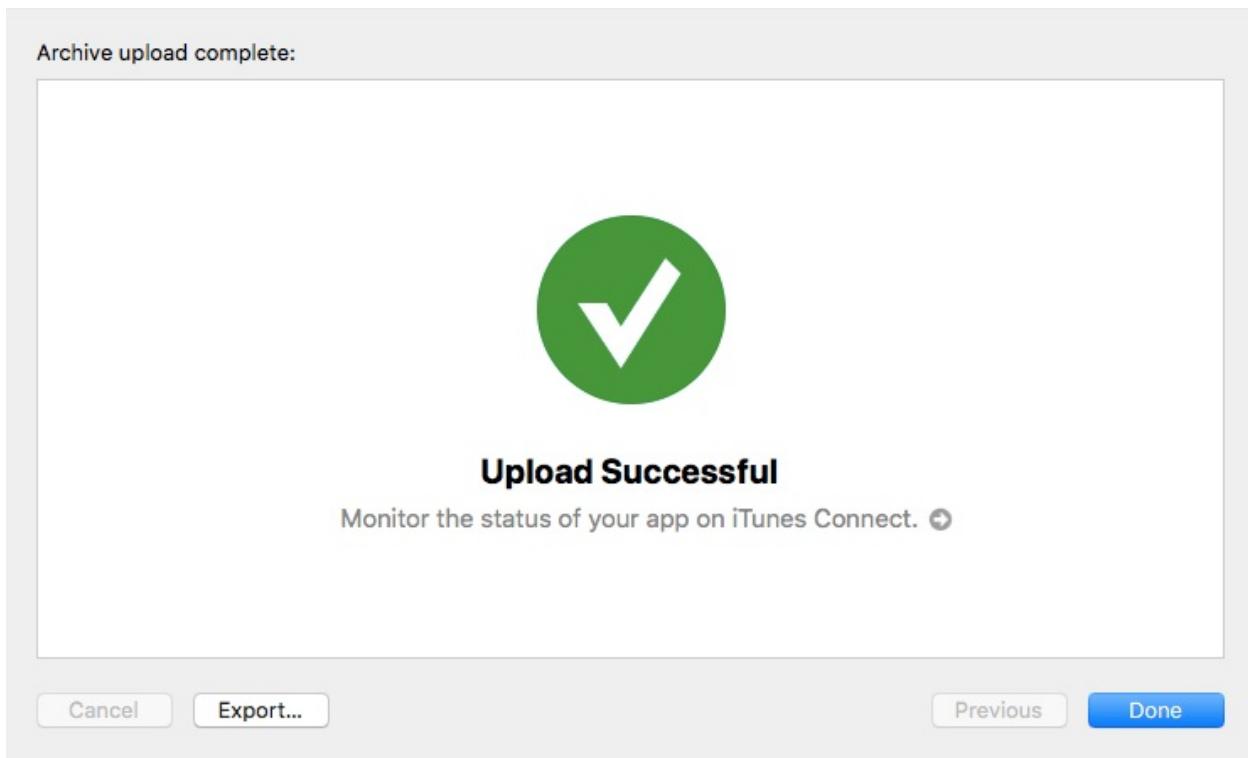
Just select, **Generate an iOS Distribution Certificate** and then choose **Next**.

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. Once you have done that click **Next**.

The next screen will allow you to review your information:

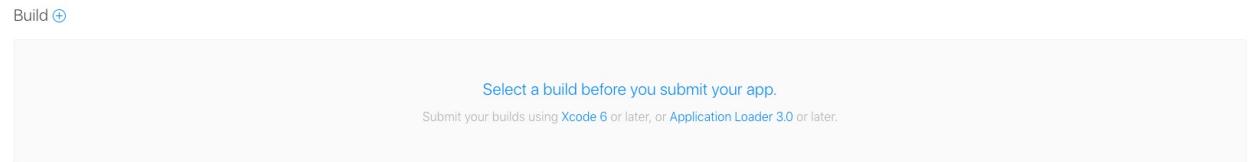


If you are happy, just hit **Upload**. Once the application has successfully uploaded you should see an upload successful screen:



Woohoo!, almost there. Once the app is uploaded, you will be able to associate it with the app record you created in [iTunes Connect](#). To do that, click on the current version of the application on the left of the screen (e.g. **1.0 Prepare for Submission**).

Go to the **Builds** section (you will need to scroll down a little). If your upload has finished processing, you should see a '+' icon next to 'Builds':



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.

Click the add icon, 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 iTunes 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 iTunes 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

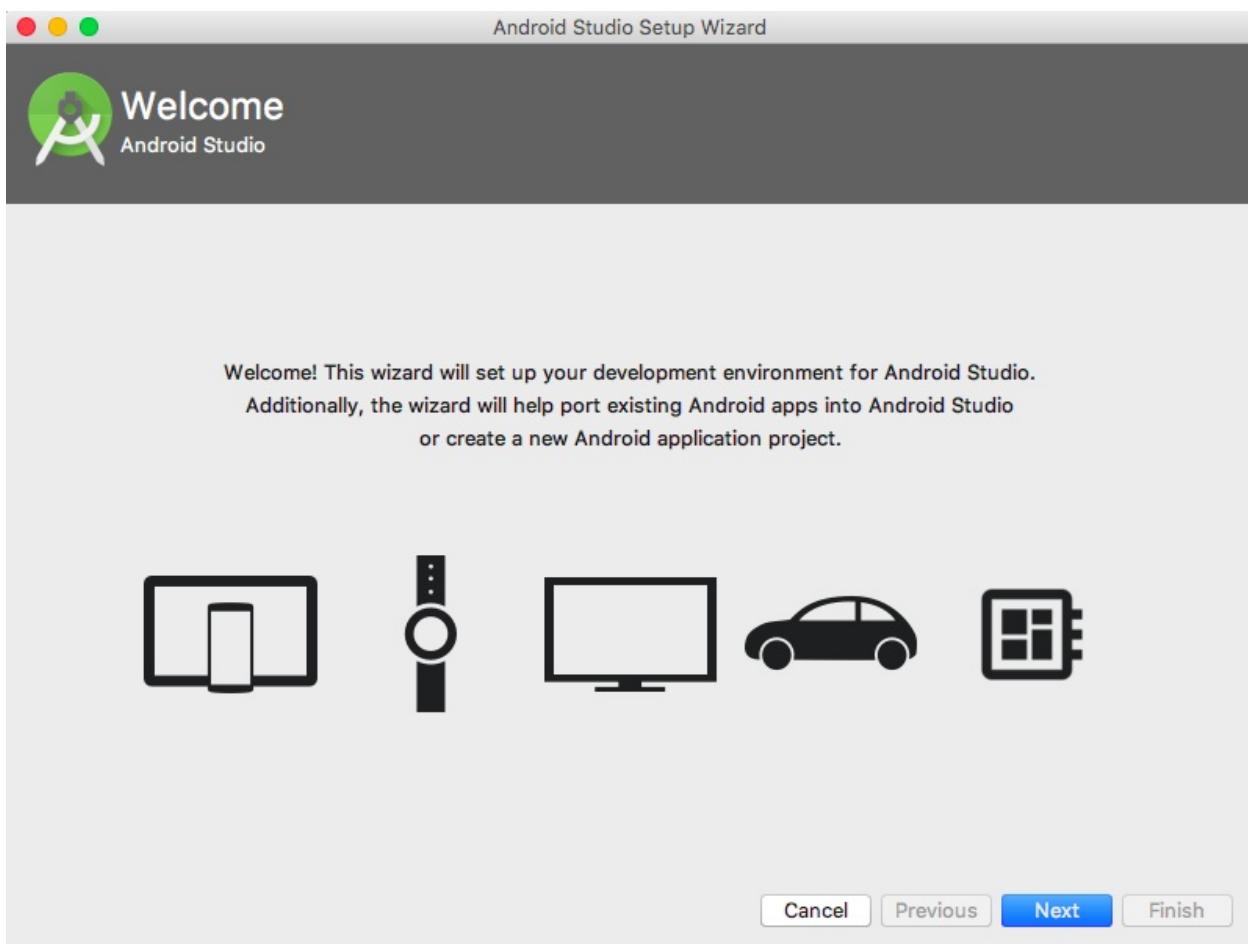
Unlike iOS development, you can build and submit Android applications no matter what operating system you are using. However, you will still need to [register for a developer account](#) and pay a \$25 USD fee (this is just a one-time fee).

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.

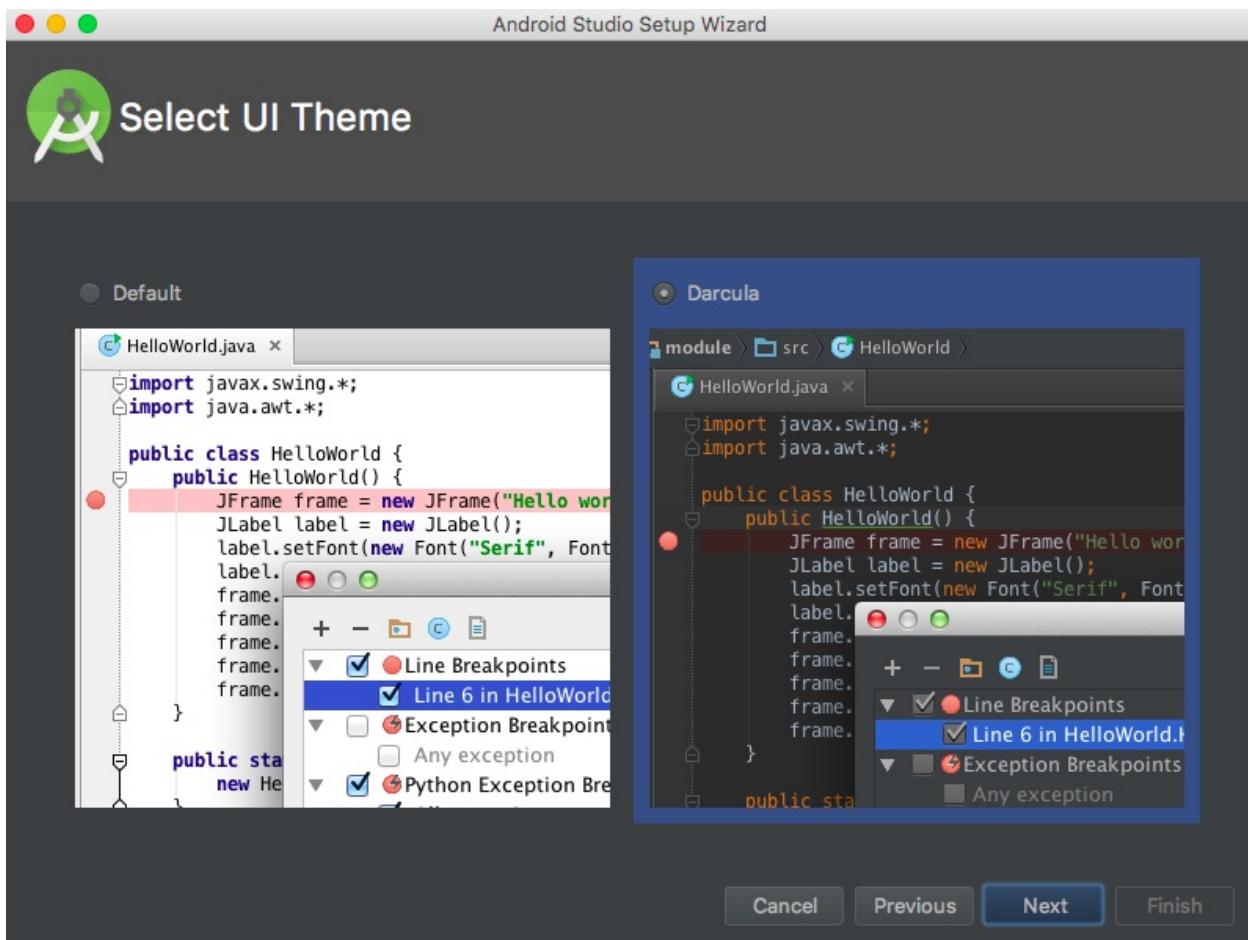
2. Install Android Studio

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](#).

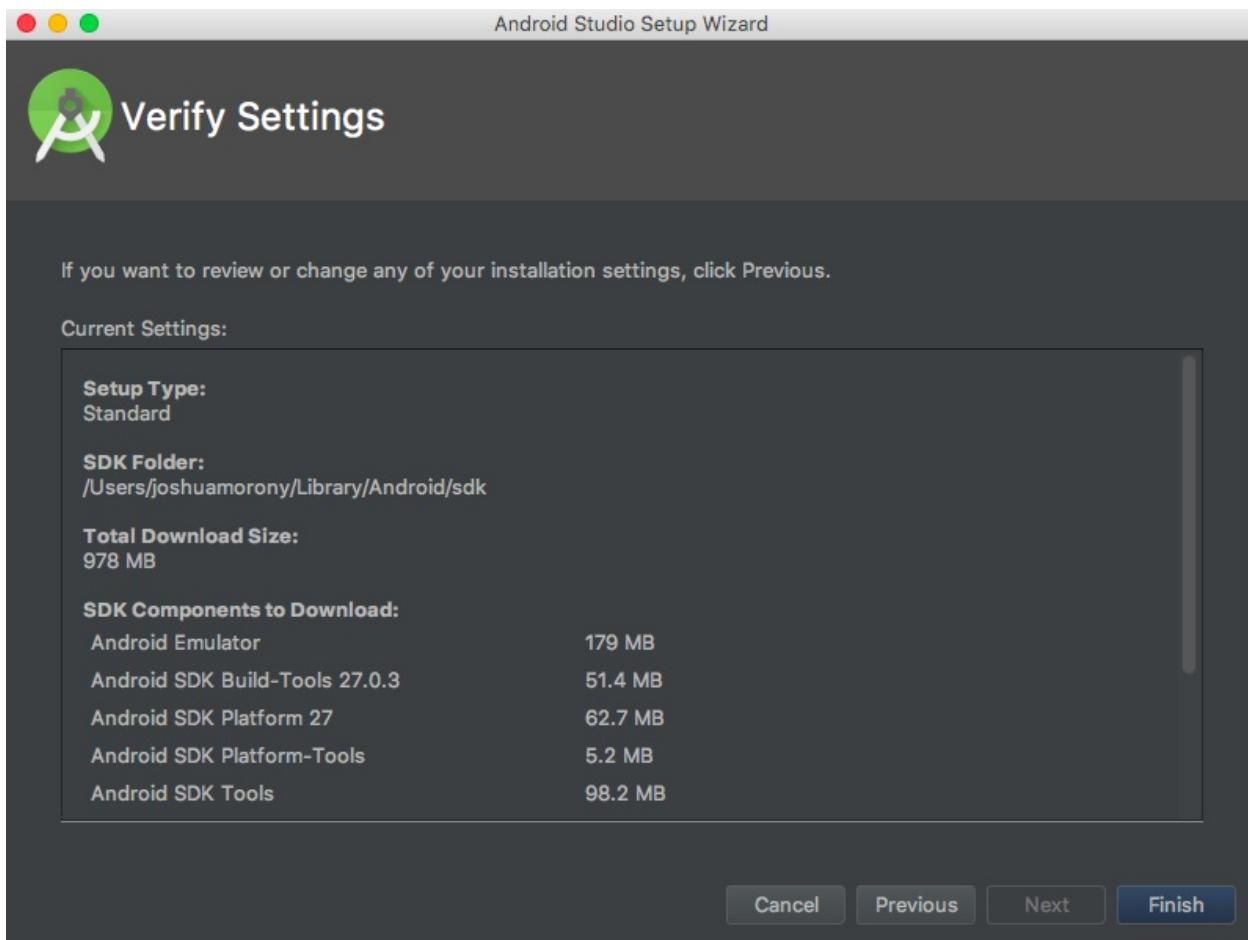
Once it is installed, you should open it. The first time you open Android Studio you will be taken through a setup wizard:



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**.

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](#) library. 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 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:

```
npm install -g n
```

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

```
n 8.11.1
```

4. Create Your Web Application

At this stage, you will need your web application that you want to deploy to Android. 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.

5. Install Capacitor

You should also have Capacitor installed at this stage, and you should have added the native platforms that you are targetting. 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 you supply to Capacitor during the configuration step (e.g. com.example.yourapp). 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
```

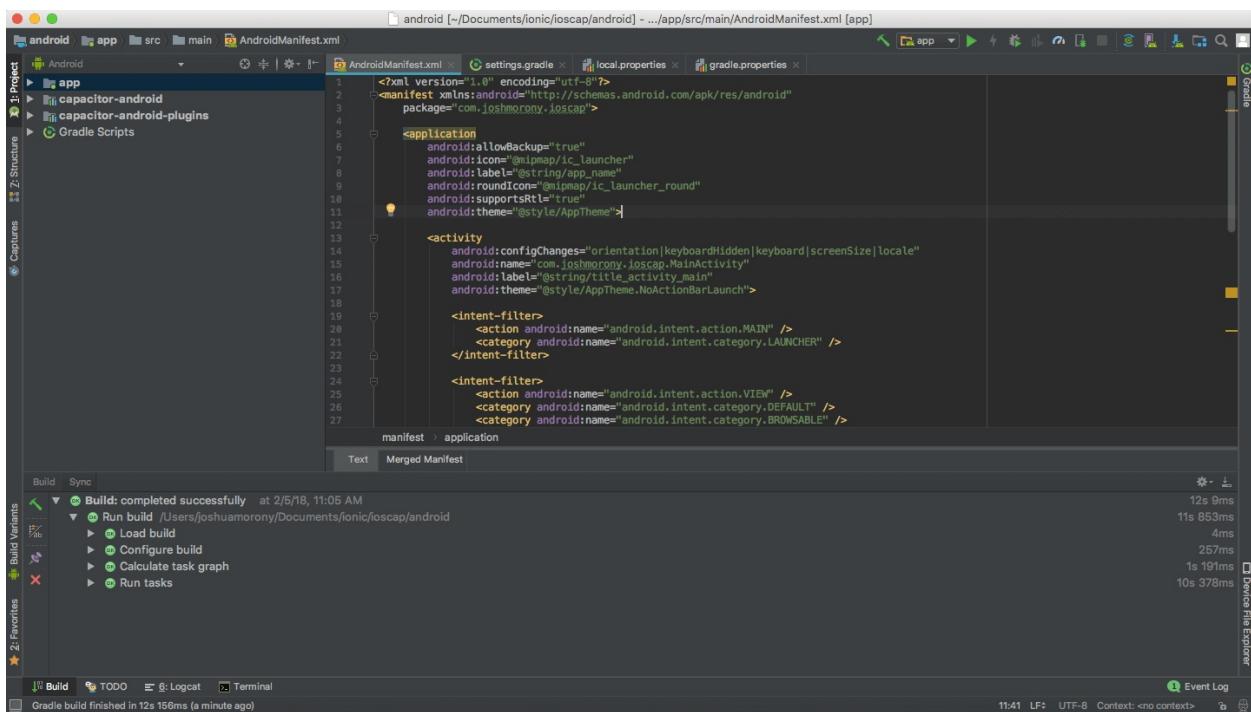
You do not need to run the sync command each time you make changes to your project. If you are only making changes to the web code of your application, you can run the following command instead:

```
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:

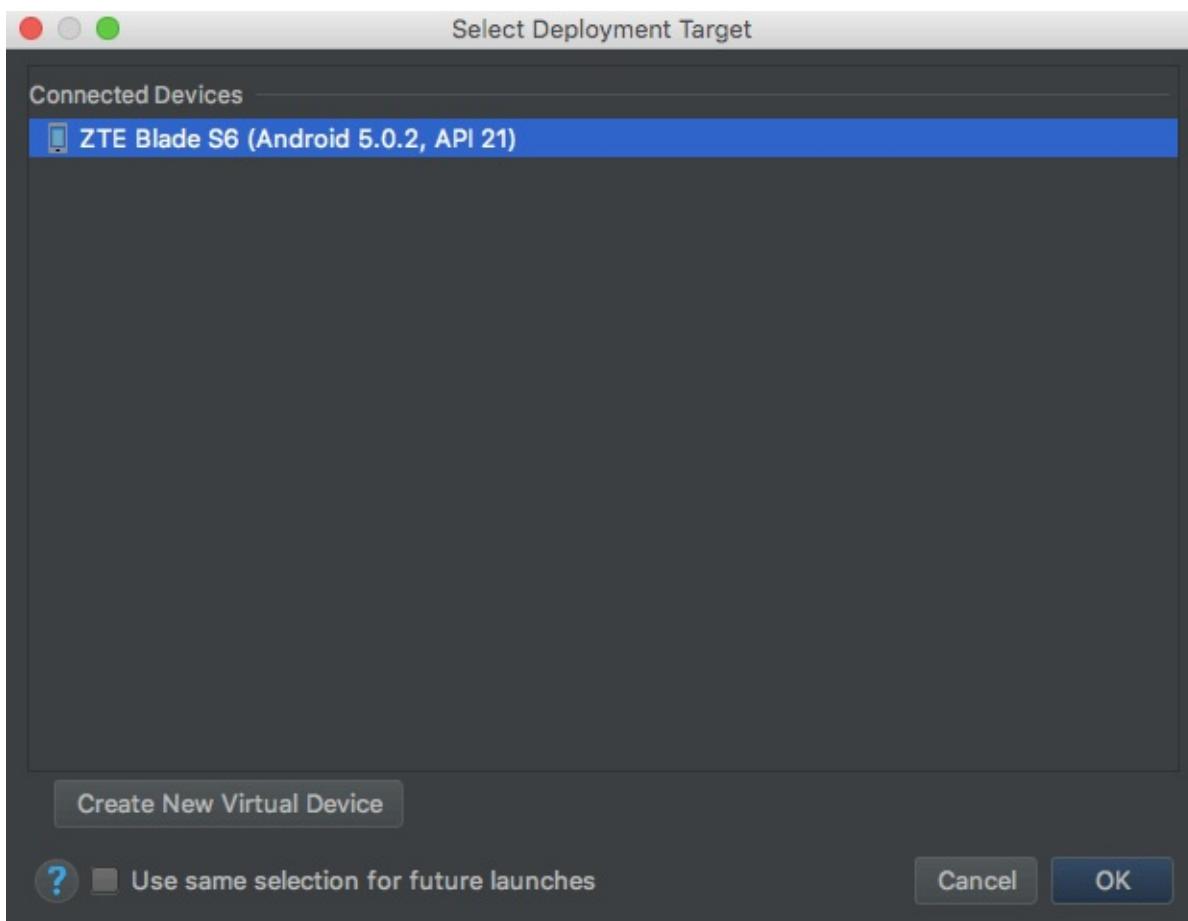


Just like with the iOS lesson, we are now dealing with a standard native Android project. The steps from here on are the same as what you would do for building and submitting any native project.

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.

6.1 Dealing with Dependency Issues

Often when building your application in Android Studio, you may run into errors like this:

```
Android dependency 'com.android.support:customtabs' has different
version for the compile (27.0.2) and runtime (27.1.1) classpath.
You should manually set the same version via DependencyResolution
```

In order to resolve these dependency issues, you will need to force the dependency to use the version specified in the error. To do that, open Gradle Scripts in Android Studio, and then open the build.gradle (Module: app) file. To solve this particular error, you would need to add:

```
implementation 'com.android.support:customtabs:27.1.1'
```

to your dependencies. That might look like this:

```
dependencies {
    implementation fileTree(include: ['*.jar'], dir: 'libs')
    implementation 'com.android.support:appcompat-v7:27.1.1'
    implementation 'com.android.support.constraint:constraint-
layout:1.0.2'
    implementation project(':capacitor-android')
```

```
testImplementation 'junit:junit:4.12'
androidTestImplementation
'com.android.support.test:runner:1.0.1'
    androidTestImplementation
'com.android.support.test.espresso:espresso-core:3.0.1'
    implementation project(':capacitor-cordova-android-plugins')
    implementation 'com.google.firebaseio:firebase-
messaging:17.1.0'
    implementation 'com.android.support:support-v4:27.1.1'
    implementation 'com.android.support:customtabs:27.1.1'
}
```

Notice that we have added the new dependency to the bottom. If you were to then get an error like this:

```
Android dependency 'com.android.support:support-v4' has different
version for the compile (27.0.2) and runtime (27.1.1) classpath.
You should manually set the same version via DependencyResolution
```

Then you would upgrade that to 27.1.1 in your dependencies in the same build.gradle file as well.

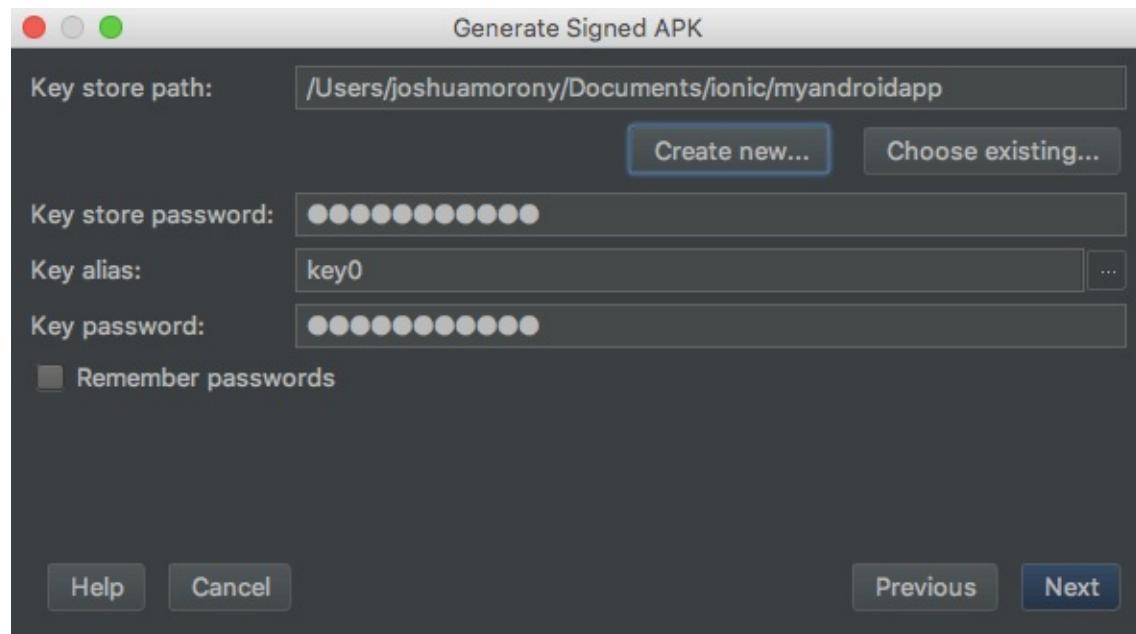
7. Build a Signed APK

Once you are ready to submit your application, you will need to build a **signed APK** with Android Studio. An APK 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.

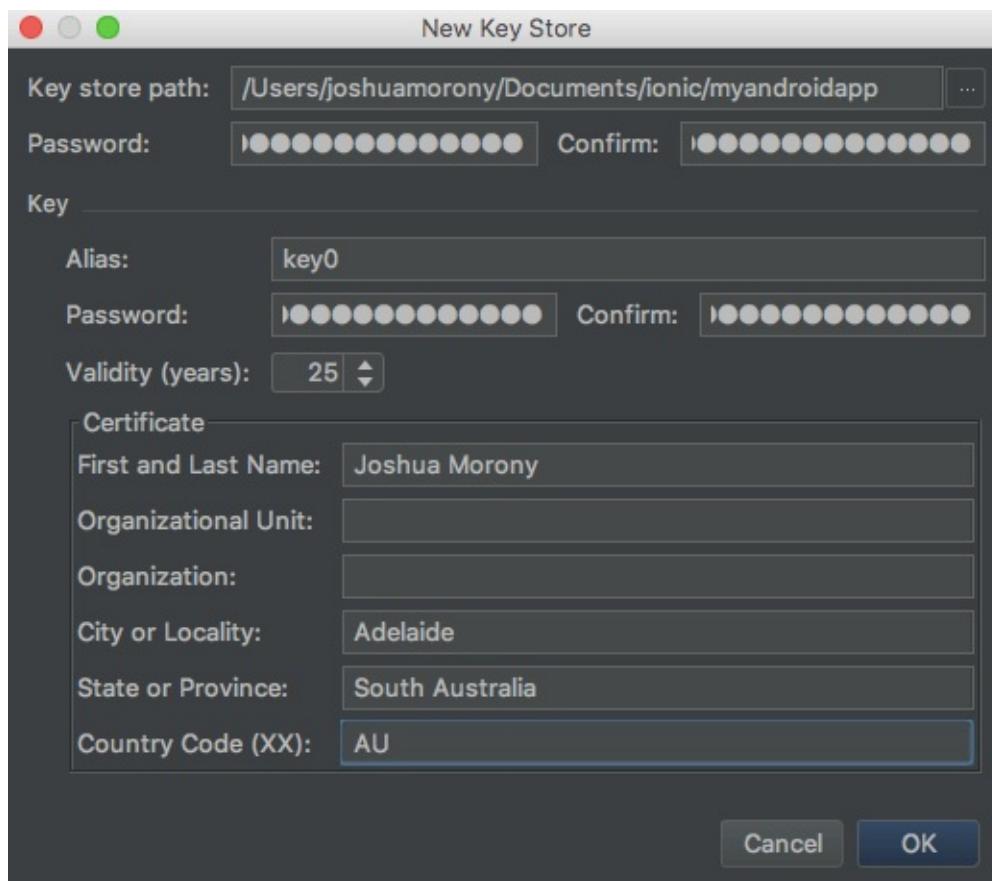
We will just be walking through the basics steps of signing and uploading an APK to Google Play. As I mentioned in the iOS lesson, make sure that the application you are distributing is a production build of your application (i.e. a minimised/optimised one).

In order to build your signed APK, you should first go to **Build > Generate Signed APK** in the menu at the top of Android Studio, then click **Next**.



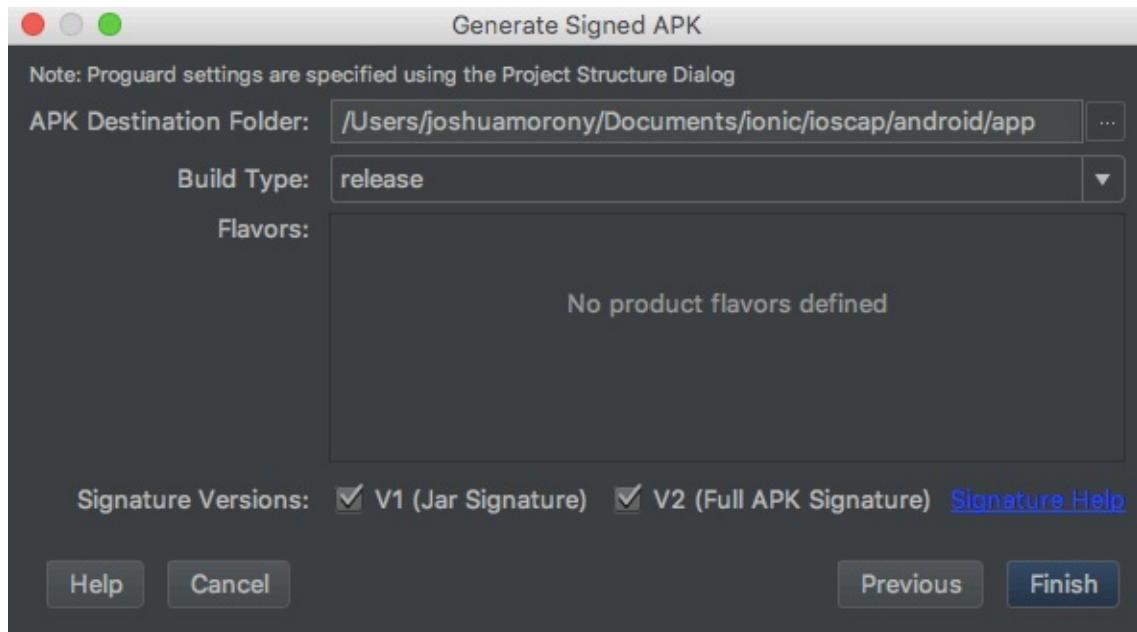
On this screen, you should supply your private key (keystore file) for this project, or if you do not yet have a keystore file you should select **Create New**.

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. When you are done, click **OK** and then **Next**.

Make sure to choose the 'release' build type, and you should also tick both of the signature options:



Once you complete this step, the application will begin building. Once it has finished you will be able to find your signed APK at:

yourproject/android/app/**release**/app-**release**.apk

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 APK file to Google Play. To begin, you should go to the [Google Play Developer Console](#). Once you have logged in, click

Create Application and give your application a name.

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 APK in this lesson.

On the left of the page, you will find a bunch of different options:



[All applications](#)

[App releases](#)



[Android Instant Apps](#)

[Artifact library](#)

[Device catalog](#)



[App signing](#)

[Store listing](#)



[Content rating](#)



[Pricing & distribution](#)



[In-app products](#)

[Translation service](#)

[Services & APIs](#)

[Optimization tips](#)



You will want to go to **App Releases**. This is where you will be able to upload your signed APK. You can use this section to upload alpha/beta test versions for your application as well, but if you are ready to submit to the Google Play store you just need to upload to the **Production** section:

Production

[MANAGE PRODUCTION](#)



Add APKs to production to make your app available to all users on the Google Play Store.

Click on **Manage Production** and then **Create Release**.

You may be prompted to use "Google Play App Signing" at this stage. Basically, Google can manage the signing of your application for you. One benefit to this is that if you lose your signing key you won't necessarily be locked out of your application forever, as Google has the power to grant you access to it. If you are more security oriented you may not find this option particularly appealing and may prefer to keep the keystore file under your own control. I've chosen to manage my own signing by opting out in this example, but you can use whichever approach you prefer.

Next, you should add your signed APK to the 'APKs to add' section, and fill out the **What's new in this release?** section:

Drop your APK file here, or select a file.

BROWSE FILES

Version code

1 REMOVE

+

i

Release name

Name to identify release in the Play Console only, such as an internal code name or build version.

1.0 3/50

Suggested name is based on version name of first APK added to this release.

What's new in this release?

Release notes translated in 0 languages

Enter the release notes for each language within the relevant tags or copy the template for offline editing. Release notes for each language should be within the 500 character limit.

```
<en-GB>
This is just a test!
</en-GB>
```

G

DISCARD

SAVE REVIEW

Hit **Save** and then **Review**. 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.

Creating iOS Certificates on Windows

In order to sign and submit your iOS application, you, unfortunately, need to use a macOS machine. However, it is still possible to create the necessary signing assets on a non-macOS machine, and there are services available that can build your app for you if you provide these assets.

In this lesson, we are going to walk through how to create a CSR (Certificate Signing Request) file, the necessary certificates, and provisioning profiles through the Apple developers portal, and the Personal Information Exchange (.p12) file that you will need to supply to sign your application. This is a reasonably long and painful process, opposed to a reasonably quick and easy process on a macOS machine, but unfortunately, it seems that's the price Apple makes you pay for not buying their products.

Install OpenSSL

To complete the following steps you will need to download and install OpenSSL. Visit the OpenSSL website:

<https://www.openssl.org/source/>

and download the version appropriate for your system.

Generate a Certificate Signing Request

Once you have OpenSSL set up we are going to use it to generate a Certificate Signing Request. This will be used in the iOS Member Center to create either our development or distribution certificate.

- Change your directory to the OpenSSL bin directory in the command prompt, e.g:

```
cd c:/OpenSSL-Win64/bin
```

- Generate a private key

```
openssl genrsa -out mykey.key 2048
```

- Use that key to generate a certificate signing request

```
openssl req -new -key mykey.key -out myCSR.certSigningRequest -  
subj "/emailAddress=you@yourdomain.com, CN=Your Name, C=AU"
```

Make sure to replace the email address, name and country code with your own.

Create a Certificate

To complete these next steps you will need to log in to the [Apple Member Center](#) and go to the **Certificates, Identifiers & Profiles** section:



SDKs

Download the SDKs and the latest beta software.

Forums

Find answers and ask questions.

Certificates, Identifiers & Profiles

Manage your certificates, identifiers, devices, and profiles for your apps.

Bug Reporting

Submit bugs or feature requests.

iTunes Connect

Manage your apps published on the App Store and Mac App Store.

Technical Support

Request technical support.

- Choose **Certificates** under *iOS Apps*
- Click the + button in the top right corner:



Next, you must choose what type of certificate you want to generate. If you are creating a certificate for testing applications then you should choose **iOS App Development** but if you are preparing an application for distribution on the App Store then you should choose **App Store and Ad Hoc**.

The screenshot shows a process flow with four steps: 'Select Type', 'Request', 'Generate', and 'Download'. The 'Select Type' step is active, indicated by a blue arrow pointing right above it. The title 'What type of certificate do you need?' is displayed above the selection options. There are two main sections: 'Development' and 'Production'. Under 'Development', the 'iOS App Development' option is selected (indicated by a blue circle) and described as 'Sign development versions of your iOS app.'. The 'Apple Push Notification service SSL (Sandbox)' option is also listed but not selected. Under 'Production', the 'App Store and Ad Hoc' option is listed but not selected. Below the 'Development' section, a note states: 'Establish connectivity between your notification server and the Apple Push Notification service sandbox environment. A separate certificate is required for each app you develop.'

Development

iOS App Development
Sign development versions of your iOS app.

Apple Push Notification service SSL (Sandbox)
Establish connectivity between your notification server and the Apple Push Notification service sandbox environment. A separate certificate is required for each app you develop.

Production

App Store and Ad Hoc
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.

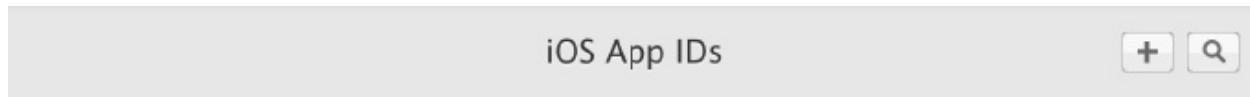
You will also notice some other options on this screen. You will need to create separate certificates if you want to use things like the Apple Push Notifications service or WatchKit, but we won't require any of those.

- Select your certificate type and click ***Continue***
- It will now ask you for the certificate signing request you just created with OpenSSL, click continue and then upload the signing request. Once you have selected the **.certSigningRequest** file click **Generate**.

You will now be able to download your certificate. Download it to somewhere safe and then open it to install it (for convenience, you should place it in your OpenSSL bin folder (e.g. **OpenSSL-Win64/bin** because this is where we will be running some commands later)).

Create an Identifier

- Click on **App IDs** and then click the + icon



- Fill in the App ID Description and then supply an Explicit App ID like the following:

App ID Suffix

Explicit App ID

If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

Wildcard App ID

This allows you to use a single App ID to match multiple apps. To create a wildcard App ID, enter an asterisk (*) as the last digit in the Bundle ID field.

Bundle ID:

Example: com.domainname.*

This should match the **id** that you supply in your **config.xml** file.

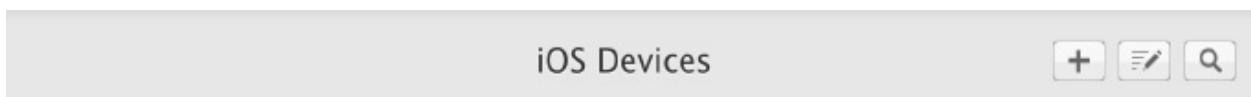
- Go to the bottom of the screen and click **Continue**
- On the following screen click **Submit**

The App ID should now be registered and available to you to use in provisioning profiles, which we will do in the next step.

Create a Provisioning Profile

Before you create a provisioning profile you will need to add devices that the application can run on.

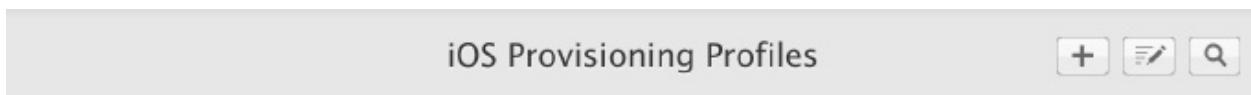
- Click on **Devices** and then click the + button in the top right:



- Supply the name and UDID (you can find this for your device in iTunes) of the device you want to register and then click **Continue**. Follow the prompts to finish.

Now we can create the provisioning profile.

- Go to the provisioning profiles screen and click the + icon in the top right



- Choose **iOS App Development** if you are creating a provisioning profile for testing, or **App Store** if you are creating a provisioning profile for distribution. Click **Continue**.
- Select the App ID you just created and click **Continue**:



Select App ID.

If you plan to use services such as Game Center, In-App Purchase, and Push Notifications, or want a Bundle ID unique to a single app, use an explicit App ID. If you want to create one provisioning profile for multiple apps or don't need a specific Bundle ID, select a wildcard App ID. Wildcard App IDs use an asterisk (*) as the last digit in the Bundle ID field. Please note that iOS App IDs and Mac App IDs cannot be used interchangeably.

App ID: ▼

- Select the certificate you wish to use and click **Continue**:



Select certificates.

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

<input type="checkbox"/> Select All	0 of 1 item(s) selected
<input type="checkbox"/> Joshua Morony (iOS Development)	

- Select all the devices you want to run the application on and click **Continue**
- Provide a name for the provisioning profile and click **Generate**



Name this profile and generate.

The name you provide will be used to identify the profile in the portal.

Profile Name:

Type: **iOS Development**

App ID: **Test App(9YABKUX5J7.com.joshmorony.test)**

Certificates: **1 Included**

Devices: **11 Included**

You should now be able to download your Provisioning Profile (again, download it into that same OpenSSL bin folder to make things easier in this next step).

Generating a .p12

Finally, to create the **.p12** file we will use the certificate we downloaded from the Member Center and a couple more commands.

- Run the following command to generate a PEM file:

```
openssl x509 -in ios_development.cer -inform DER -out  
app.pem_file.pem -outform PEM
```

- Use the key you generated right back at the beginning and the PEM file you just created to create your .p12 file:

```
openssl pkcs12 -export -inkey mykey.key -in app_pem_file.pem -out app_p12.p12
```

There are a lot of things that can go wrong here and it's easy to mess up. So if you have any trouble I would suggest just restarting the whole process and slowly and carefully make sure you type everything out correctly.

Congrats! You now have your **.p12** file. Since you have both your provisioning profile and **.p12** file now, if you are using a service that builds your app for you (like Ionic Package) you will be able to attach these files to your application when you upload it. Once you have done that you should be able to generate a signed **.ipa** file that can be installed on your device (as long as you added your device and have followed all these steps correctly!).

Building for iOS and Android Using Ionic Package (Coming Soon)

Ionic Package is a service provided by Ionic that will handle building your applications for you, so you don't need to worry about dealing with the native SDKs on your computer. The primary benefit of Ionic Package is to people who want to develop iOS applications, but do not have access to a macOS machine. The macOS operating system is required in order to compile native iOS applications.

This service is not currently available for Capacitor applications, but support is currently planned. This lesson will be updated once support is available in Ionic Package for Capacitor applications.

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 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 deserve 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 it. 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 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.

Keep in mind that service workers are a relatively new technology, and are not universally supported in browsers. Service workers have been supported by Chrome for quite a while already (which would allow your application to run as a PWA on an Android device), but iOS/Safari has been dragging its feet for quite some time. Fortunately, recent updates to iOS Safari have included support for service workers. For a list of where service workers are currently supported, you can view [this page](#).

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.

Once you run this command, you will see various files being created and updated:

```
Installed packages for tooling via npm.

CREATE ngsw-config.json (392 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)
CREATE src/manifest.json (1063 bytes)
UPDATE angular.json (4093 bytes)
UPDATE package.json (1760 bytes)
UPDATE src/app/app.module.ts (993 bytes)
UPDATE src/index.html (759 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:

```
{
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/*.css",
          "/*.js"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "resources": {
        "files": [
          "/*"
        ]
      }
    }
  ]
}
```

```
"updateMode": "prefetch",
"resources": {
  "files": [
    "/assets/**"
  ]
}
}]
```

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
```

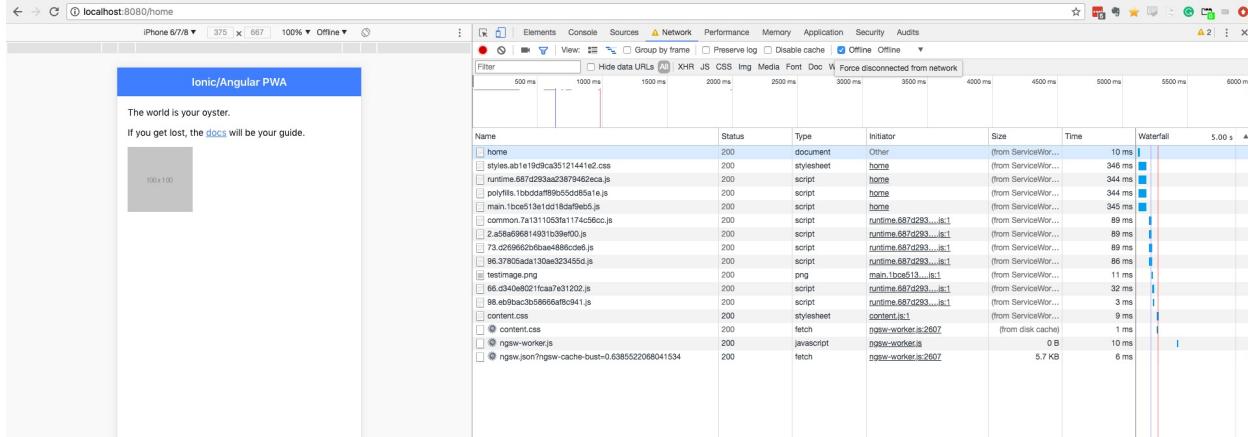
and then serve the contents of that folder:

```
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 had 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.

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.

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 1GB of storage and 10GB 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 (if you aren't familiar with the process, configuring SSL certificates is a pain). 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.firebaseio.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.

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**.

You should modify your **firebase.json** file to look like this (if it doesn't look like this already - this is the file generated in the `init` process):

```
{  
  "hosting": {  
    "public": "www",  
    "ignore": [  
      "firebase.json",  
      "**/*.*",  
      "**/node_modules/**"  
    ],  
    "rewrites": [  
      {
```

```
        "source": "**",
        "destination": "/index.html"
    },
]
}
}
```

and the `.firebasesrc` file just defines the project that this application is associated with.

With all of that done, all you need to do is run:

```
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.json 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).## Conclusion

If you've made it through this course (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 HTML5 mobile application development (which I think is a very wise choice), you've also invested time and money into me. I love teaching developers HTML5 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 course then you're probably familiar with [my blog](#), but if you're not make sure to check it out. I release free HTML5 mobile development tutorials there every week, and if you're looking to expand your Ionic skills even more there's plenty to check out.

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.

I hope you enjoyed this course 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 HTML5 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 fellow HTML5 mobile developers are up to!