JOSHUA MORONY

BUILDING MOBILE APPS WITH IONIC 2

This is a PREVIEW copy. It includes some of the introductory sections, as well as a single lesson from the "QuickLists" application. To give you a better idea of what content different sections of the book contain, the content in the preview version does not flow from one section to the next, it instead takes samples from various sections in the book.

To purchase the full book, please click here.

Contents

Chapter 1

Introduction

Welcome!

Hello and welcome to **Building Mobile Apps with Ionic 2!** This book will teach you everything you need to know about Ionic 2, 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 when reading this book, many of you will already be familiar with Ionic 1, some may have some experience with Ionic 2, and some may have no experience with either. 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. If there are concepts that need to be explained in more than one walkthrough, I have copied information into both rather than referring to the other walkthrough.

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 is more of a logical progression to watch it first.

Updates & Errata

Ionic 2 is still in development, so that means that it is still changing. It is reasonably stable now, so most of what you read in this book won't change, but there will still most likely be some changes until the release version is reached. I will be frequently updating this book to reflect any changes that are made to the framework, 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 get an update out as soon as I can.

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 walk throughs. 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!

New Concepts

Ionic 1 was built on top of Angular 1, which is a framework for building complex and scaleable Javascript applications. What Ionic does on top of Angular is that it provides a bunch of functionality to make making mobile apps with Angular easier. Then along came Angular 2 which is the next iteration of the Angular framework, which comes with a bunch of changes and improvements. In order for Ionic to make use of Angular 2 a new framework was required on their end as well, which is how Ionic 2 came about. In short, by using Ionic 2 & Angular 2 we will be able to make apps that perform even better on mobile, adhere to the latest web standards, are scalable, reusable, modular and so on.

With the introduction of Angular 2, there has been a lot of changes to how you develop an application. There are massive conceptual changes, and there have also been a few changes to things like template syntax as well.

In Ionic 2, your templates will look something like this:

```
<ion-menu [content]="content">

  <ion-toolbar>
    <ion-title>Pages</ion-title>
  </ion-toolbar>

  <ion-content>
    <ion-list>
      <button ion-item *ngFor="let p of pages" (click)="openPage(p)"></button>
    </ion-list>
  </ion-content>

</ion-menu>

<ion-nav id="nav" [root]="rootPage" #content></ion-nav>
```

which isn't too different to Ionic 1, and your Javascript will look something like this:

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { HomePage } from './pages/home/home';

@Component({
  template: `<ion-nav [root]="rootPage"></ion-nav>`
})
export class MyApp {

  rootPage: any = HomePage;

  constructor(platform: Platform) {

    platform.ready().then(() => {

    });
  }
}
```

which is *very* different to Ionic 1. If you're already familiar with ECMAScript 6 or TypeScript then a lot of this probably won't be too hard of a change for you, but if these are completely new concepts to you (and for most people it will be) the transition might be a little more difficult. To help put your mind at ease somewhat, ES6 and TypeScript was all completely new to me when the Ionic 2 alpha first came out, and within a pretty short time period, I started to feel very comfortable with it. Now I am way more comfortable with the new syntax and structure than I ever was with Ionic 1.

In this lesson we are going to broadly cover some of the new concepts and syntax in Ionic 2 & Angular 2. The intention is just to give you a bit of a background, we will get into specifics later.

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 will bring about some pretty drastic changes to the way you will 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.

Modules

```
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2PI = " + math.sum(math.pi, math.pi))

//  otherApp.js
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 more later, but essentially any components we create in our application we "export" so that we can "import" them elsewhere.

Promises

Promises are something that have been made available by services like ngCordova previously, but now they are natively supported, meaning you can do something like this:

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

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 allow 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.

Fat Arrow Functions

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.

This is by no means an exhaustive list of new ES6 features so for some more examples take a look at es6-features.org.

TypeScript

Another concept we should cover off on is TypeScript which is used in Ionic 2. It's important to point out that although Ionic 2 uses TypeScript, you don't have to use it yourself to build Ionic 2 applications - you can just use plain ES6. That said though, TypeScript provides additional features and makes some things (dependency injection in particular) a lot easier, and it will soon become the default for Ionic 2 so it doesn't make much sense not to use it.

We will be using TypeScript in this book, so let's talk a little bit more 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 (it seems easy to understand definitions are a big no-no in the tech world). In fact, a StackOverflow post did a much better job at explaining what TypeScript is – basically, TypeScript adds typing, classes and interfaces to JavaScript.

Using TypeScript allows you to program in the way you would for stricter, object oriented languages like Java or C#. Javascript wasn't originally intended to be used for designing complex applications so the language wasn't designed that way. It certainly is possible already to use JavaScript in an object oriented manner by using functions as classes as we discussed before but it's not quite as clean as it could be.

But… I mentioned before that ES6 is already adding the ability to create classes so why do we still need TypeScript? I saw one Redditor put it quite simply:

"It's called TypeScript not ClassScript"

TypeScript still provides the ability to use static typing in JavaScript (which means it is evaluated at compile time, opposed to dynamic typing which is evaluated at run time). Using typing in TypeScript will look a little like this:

```
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 you take a look at the Ionic 2 code from before:

```
export class MyApp {

  rootPage: any = HomePage;

  constructor(platform: Platform) {

    platform.ready().then(() => {

    });
  }
}
```

You can see some TypeScript action going on. The code above is saying that `rootPage` can be the `any` type, which is a special type which basically just means it can be anything at all, and `platform` has a type

of `Platform`. As you will see later, the ability to give things types comes in very handy for an important concept called **dependency injection**.

Since the default option for Ionic 2 is TypeScript, and it is what most people are using, this book focuses on using TypeScript. For the most part, ES6 and TypeScript projects look pretty much the same, and converting between the two is a reasonably straight forward task.

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 automatically handled by Ionic.

Web Components

Web Components are kind of the big thing in Angular 2, and they weren't really feasible to use in Angular 1. Web Components are not specific to Angular, they are becoming a new standard on the web to create modular, self contained, pieces of code that can easily be inserted into a web page (kind of like Widgets in WordPress).

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

Rob Dodson wrote a great post on Web Components where he explains how they work and the concepts

behind it. He also provides a really great example, and I think it really drives the point home of why Web Components are useful.

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

```
<img-slider>
  <img src="images/sunset.jpg" alt="a dramatic sunset">
  <img src="images/arch.jpg" alt="a rock arch">
  <img src="images/grooves.jpg" alt="some neat grooves">
  <img src="images/rock.jpg" alt="an interesting rock">
</img-slider>
```

instead of (without web components) this:

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

```
  <label for="slide4"></label>
</div>
```

In the future, 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.

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.

Chapter 2

Ionic 2 Basics

Lesson 1: Generating an Ionic 2 Application

We've covered quite a bit of context already, so you should have a reasonable idea of what Ionic 2 is all about and why some of the changes have been made. With that in mind, we're ready to jump in and start learning how to actually use Ionic 2.

Installing Ionic

Before we can start building an application with Ionic 2 we need to get everything set up on our computer first. It doesn't matter if you have a Mac or PC, you will still be able to finish this book and produce both an iOS and Android application that is ready to be submitted to app stores.

IMPORTANT: If you already have Ionic 1 set up on your machine then you can skip straight to the next section. All you will need to do is run `npm install -g ionic` or `sudo npm install -g ionic` to get everything needed for Ionic 2 set up. Don't worry if you want to keep using Ionic 1 as well, after you update you will be able to create both Ionic 1 and Ionic 2 projects.

First you will need to install Node.js on your machine. Node.js is a platform for building fast, scalable network applications and it can be used to do a lot of different things. Don't worry if you're not familiar with it though, we won't really be using it much at all - we need it installed for Ionic to run properly and to install some packages but we barely have to do anything with it.

> Visit the following website to install Node.js:

https://nodejs.org/

Once you have Node.js installed, you will be able to access the node package manager or npm through your command terminal.

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

```
npm install -g ionic cordova
```

or

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

You should also set up the Android SDK on your machine by following one of these guides:

- Installation for Mac
- Installation for Windows

If you are on a Mac computer then you should also install XCode which will allow you to build and sign applications.

You don't have to worry about setting up the iOS SDK as if you have a Mac this will be handled by XCode and if you don't have a Mac then you can't set it up on your computer anyway (we'll talk more about how you can build iOS applications without a Mac later).

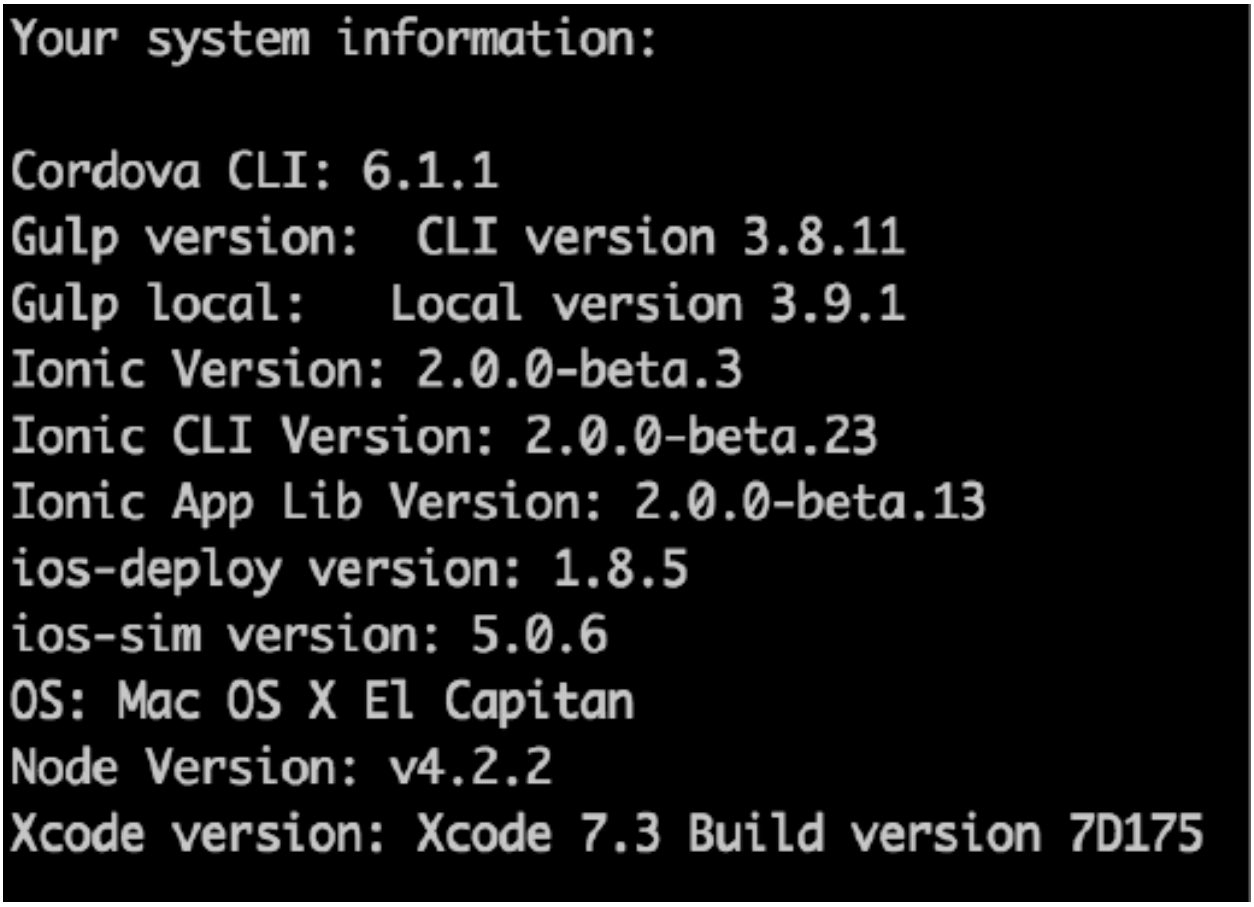
You should now have everything you need set up and ready to use on your machine! To verify that the Ionic CLI (Command Line Interface) is in fact installed on your computer, run the following command:

```
ionic -v
```

You can also get some detailed information about your current installation by running the following command from within an Ionic project:

```
ionic info
```

It should spit out some info about your current environment, here's mine at the time of writing this:



If you run into any trouble installing Ionic or generating new projects, make sure that you have the latest (current) Node version installed. After you have the latest version installed, you should also run the following commands:

```
npm uninstall -g ionic npm cache clean
```

before attempting to install again.

NOTE: The Ionic Framework and Ionic CLI (Command Line Interface) are two separate things. The CLI is what we just installed, and it provides a bunch of tools through the command line to help create and manage your Ionic projects. The Ionic CLI will handle downloading the actual Ionic Framework onto your machine for each project you create.

Generating Your First Project

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.

> Run the following command to generate a new Ionic application:

```
ionic start MyFirstApp blank --v2
```

To generate a new application called 'MyFirstApp' that uses the "blank" template. Ionic comes with some templates built in, in the example above we are using the 'blank' template, but you could also use:

```
ionic start MyFirstApp sidemenu --v2
```

or

```
ionic start MyFirstApp tutorial --v2
```

or you could just run the default command:

```
ionic start MyFirstApp --v2
```

to use the default starter which is a tabs application. Notice that every time we are supplying the –v2 flag. If you leave this flag off it will just create a normal Ionic 1 application (handy for those of you who still need to use V1 as well, but make sure you don't forget it when building Ionic 2 apps!).

NOTE: All Ionic 2 projects use TypeScript by default now. Since TypeScript is an extension of ES6, ES6 code will still work in TypeScript projects if you want to use it, but all Javascript files should have the .ts extension, not .js.

We're just going to stick with a boring blank template for now. Once your application has been generated you will want to make it your current directory so we can do some more stuff to it.

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

```
cd MyFirstApp
```

If using the command prompt or terminal is new to you, you might want to read this tutorial for a little more in depth explanation - the content is specifically for Ionic 1 but it should give you a general sense of how the command line interface works.

Adding Platforms

Eventually we will be building our application with Cordova (in fact the application that the Ionic CLI generates is a Cordova application), and to do that we need to add the platforms we are building for. To add the Android platform you can run the following command:

```
ionic platform add android
```

and to add the iOS platform you can run:

```
ionic platform add ios
```

If you are building for both platforms then you should run both commands. This will set up your application so that it can be built for these platforms, but it won't really have any effect on how you build the application. As I will explain shortly, most of our coding will be done inside of the **app** folder, but you will also find another folder in your project called **platforms** - this is where all of the configuration for specific platforms live. We're going to talk about all that stuff way later though.

Running the Application

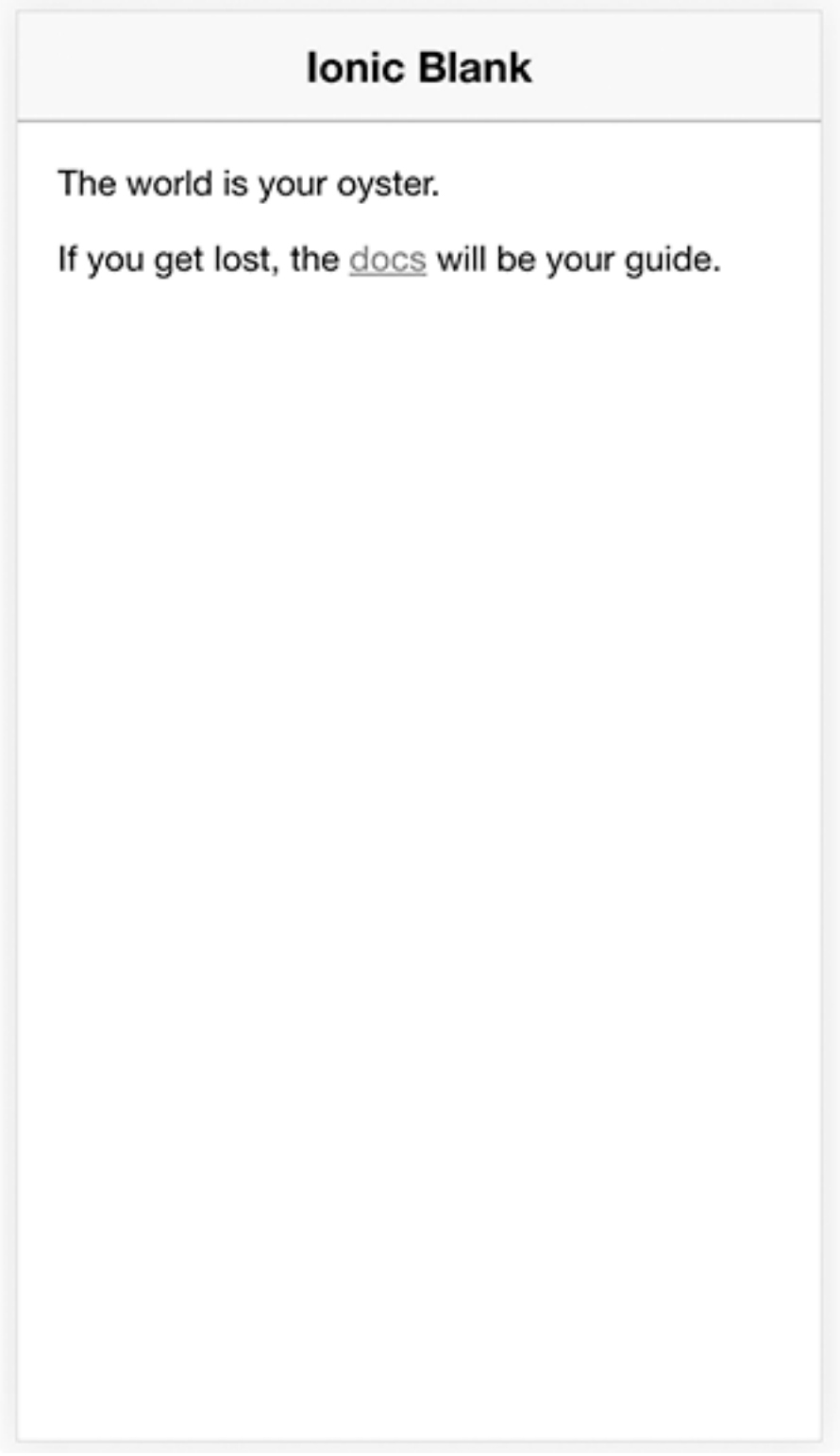
The beauty of HTML5 mobile applications is that you can run them right in your browser whilst you are developing them. But if you try just opening up your project in a browser by going to the **index.html** file location you won't have a very good time.

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:



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 later version of Ionic. 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
```

again, and then updating the **package.json** file of your project. You should see something like this in that file:

```
"dependencies": {
  "@angular/common": "^2.0.0",
  "@angular/compiler": "^2.0.0",
  "@angular/compiler-cli": "0.6.2",
  "@angular/core": "^2.0.0",
  "@angular/forms": "^2.0.0",
  "@angular/http": "^2.0.0",
  "@angular/platform-browser": "^2.0.0",
  "@angular/platform-browser-dynamic": "^2.0.0",
  "@angular/platform-server": "^2.0.0",
  "@ionic/storage": "^1.0.3",
```

```
    "ionic-angular": "^2.0.0-rc.1",
    "ionic-native": "^2.2.3",
    "ionicons": "^3.0.0",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.21"
  },
  "devDependencies": {
    "@ionic/app-scripts": "^0.0.33",
    "typescript": "^2.0.3"
  },
```

Simply change the `ionic-angular` version number to the latest version, and then run:

```
npm install
```

inside of your project directory. This will grab the latest version of the framework and add it to your project.

IMPORTANT: Keep in mind that there may be other dependencies in **package.json** that need to be updated, as well as the Ionic library.

Make sure to read the changelog to check for any breaking changes when a new version is released, which may mean that you have to update parts of your code as well.

Often it is easiest to just create a fresh new project after updating the Ionic CLI, and porting your code over. If that is not an option, just make sure you read the changelog carefully, and update your dependencies and code accordingly. As Ionic 2 becomes more and more stable, this becomes less of a problem as the changes are not as drastic.

Lesson 4: Decorators

Each class (which we will talk about in the next section) you see in an Ionic 2 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 2 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*/
}
```

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 2 application.

The decorator name itself is quite useful, here's a few you might see in an Ionic 2 application:

- @Component
- @Pipe
- @Directive

We can supply an object to the decorator to provide even more information on what we want. Here's the most common example you'll see in your applications:

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

}
```

Now this class knows where it needs to fetch its template from, which will determine what the user will actually see on the screen (we'll be getting into that later as well). 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({
    template: `<p>Howdy!</p>`
})
export class HowdyPage {

}
```

Some people even like to define large templates using `template`. 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 2 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 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 2. 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 **app** folder, which look like this:

home

- home.ts
- home.html
- home.scss

A **@Component** is not specific to Ionic 2, it is used generally in Angular 2. A lot of the functionality provided by Ionic 2 is done through using components. In Ionic 2 for example you might want to create a search bar, which you could do using one of the components that Ionic 2 provides like this:

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

You simply add this custom tag to your template. Ionic 2 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>
```

NOTE: Technically speaking a component should have a class definition and a template. Things like pipes and providers aren't viewed on the screen so have no associated template, they just provide some addi-

tional 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. 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 component use a **component**.

@Pipe

@Pipe allows you to create your own custom pipes to filter data that is displayed to the user, which can be very handy. 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.

@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 your classes, 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()
export class DataService {


}
```

IMPORTANT: Remember that just about everything you want to use in Ionic 2 needs to be imported first (we will cover importing in more detail in the next section). In the case of pipes, directives, injectables and components they not only need to be imported, but also declared in your **app.module.ts** file. We will get into the specifics around this when we go through the application examples.

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 2, but in the beginning, the vast majority of your decorators will probably just look like this:

```
@Component({
  selector: 'home-page',
```

```
  templateUrl: 'home.html'
})
```

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.

Chapter 3

Quick Lists

Lesson 4: Data Models and Observables

In this lesson we're going to design a data model for the checklists that we will use in the application, which will also incorporate Observables. A data model is not something that is specific to Ionic 2, a model in programming is a generic concept. Depending on the context, the exact definition of a model may vary, but in general a model is used to store or represent data.

In the context of Ionic 2 & Angular 2, if we wanted to keep a reference to some data we might do something like this:

```
this.myDataArray = ['1', '2', '3'];
```

However, if we were to create a model it might look something like this:

```
this.myDataArray = [
    new MyDataModel('1'),
    new MyDataModel('2'),
    new MyDataModel('3')
];
```

So instead of storing plain data, we are creating an **object** that holds that data instead. At first it might be hard to see why we would want to do this, for simple data like the example above it just looks a lot more complicated, but it does provide a lot of benefits. The main benefit for us in this application will be that it:

- Allows us to clearly define the structure of our data
- Allows us to create helper functions on the data model to manipulate our data
- Allows us to reuse the data model in multiple places, simplifying our code

Hopefully this lesson will show you how useful creating a data model can be, but let me preface this by saying this isn't something that is absolutely required. You can quite easily just define some data directly in your class if you like.

We're also going to be creating and making use of our own **Observable** in this data model, but let's cross that bridge when we get there.

Creating a Data Model

Usually if we wanted to create a data model we would create a class that defines it (it's basically just a normal object), along with its helper functions, like this:

```
class PersonModel {

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

    increaseAge(){
        this.age++;
    }

    changeName(name){
        this.name = name;
    }

}
```

Then we could create any number of instances (objects) from it like this:

```
let person1 = new PersonModel('Jason', 43);
let person2 = new PersonModel('Louise', 22);
```

and we can call the helper functions on any individual instance (object) like this:

```
person1.increaseAge();
```

The idea in Ionic 2 is pretty much exactly the same, except to do it in the Ionic 2 / Angular 2 way we create an **Injectable** (which we discussed in the basics section). Remember that an **Injectable** is used to create

services that can be injected into any of our other components, so if we want to use the data model we create we can just inject it anywhere that we want to use it.

Let's take a look at what the data model will actually look like, and then walk through the code.

> Modify `src/models/checklist-model.ts` to reflect the following:

```
export class ChecklistModel {

  checklist: any;
  checklistObserver: any;

  constructor(public title: string, public items: any[]){

    this.items = items;

  }

  addItem(item): void {

    this.items.push({
      title: item,
      checked: false
    });

  }

  removeItem(item): void {

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

    if(index > -1){
```

```
      this.items.splice(index, 1);
    }

  }

  renameItem(item, title): void {

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

    if(index > -1){
      this.items[index].title = title;
    }

  }

  setTitle(title): void {
    this.title = title;
  }

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

}
```

What we're trying to do with this data model is essentially create a blueprint for what an individual checklist *is*. A checklist has a title and it can have any number of items associated with it that need to be completed. So we set up member variables to hold these values: a simple string for the title, and an array for the items.

Notice that we allow the title and the items to be passed in through the constructor. A title must be supplied to create a new checklist, but providing an array of items is optional. If we want to immediately add items

to a checklist we can supply an items array when we instantiate it, otherwise it will just be initialised with an empty array.

We include a bunch of helper functions which are all pretty straight forward, they allow us to either change the title of the checklist, or modify any of the checklists items (by changing their name, removing an item, adding a new item to the checklist, or toggling the completion state of an item).

Also notice that we have added `: void` after each of the functions. Just like we can declare that a variable has a certain `type` by doing something like this:

```
checklist: any;
```

we can also declare what type of data a function returns. In this case, no data is being returned so we use `void`. If one of these functions were to return a string, then we would instead use `: string` on the function.

With all of that set up, we can easily create a new checklist in any component where we have imported the Checklist Model (which we will be doing in the next lesson) by using the following code:

```
let newChecklist = new ChecklistModel('My Checklist', []);
```

or

```
let newChecklist = new ChecklistModel('My Checklist', myItemsArray);
```

We're going to get a little bit fancier now and incorporate an **Observable** into our data model so that we can tell when any checklist has been modified (which will allow us to trigger a save to memory later).

Adding an Observable

You've had a little bit of exposure to Observables already in the basics section of this course - to refresh your memory we can use the Observable the **Http** service returns like this:

```
    this.http.get('https://www.reddit.com/r/gifs/new/.json?limit=10').map(res
        => res.json()).subscribe(data => {
```

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

We call the `get` method, and then `subscribe` to the **Observable** it returns. Remember that 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, since in most cases we are just retrieving the data once. The Observable is also already created for us in the case of Http.

We are about to create our very own Observable from scratch in our data model, which will allow other parts of our application to listen for when changes occur to our checklist (because we will emit some data every time a change occurs). 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 get to implementing it, 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:

```
this.http.get(url).subscribe(data => {
    console.log(data);
});
```

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:

```
this.http.get(url).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.

> Modify `src/models/checklist-model.ts` to reflect the following:

```
import {Observable} from 'rxjs/Observable';

export class ChecklistModel {

  checklist: any;
  checklistObserver: any;

  constructor(public title: string, public items: any[]){

    this.items = items;

    this.checklist = Observable.create(observer => {
      this.checklistObserver = observer;
```

```
  });

}

addItem(item): void {

  this.items.push({
    title: item,
    checked: false
  });

  this.checklistObserver.next(true);

}

removeItem(item): void {

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

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

  this.checklistObserver.next(true);

}

renameItem(item, title): void {

  let index = this.items.indexOf(item);
```

```
    if(index > -1){
      this.items[index].title = title;
    }

    this.checklistObserver.next(true);

  }

  setTitle(title): void {
    this.title = title;
    this.checklistObserver.next(true);
  }

  toggleItem(item): void {
    item.checked = !item.checked;
    this.checklistObserver.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.checklist = Observable.create(observer => {
      this.checklistObserver = observer;
    });
```

Our `this.checklist` member variable in the code above is now our very own observable. Since it is an observable, we can subscribe to it, and since it is part of our data model, we can subscribe to it on any checklist we have created in our application. For example:

```
let newChecklist = new ChecklistModel('My Checklist', []);


newChecklist.checklist.subscribe(data => {
    console.log(data);
});
```

Of course, we aren't doing anything with the Observable yet so it's never going to trigger that `onNext` response. This is why we have added the following bits of code to each of our helper functions:

```
this.checklistObserver.next(true);
```

So whenever we use one of our helper functions to change the title, or add a new item, or anything else, it will notify anything that is subscribed to its Observable. All we want to know 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.

The result of this is that now we can "observe" any checklists we create for changes that occur. Later on we will make use of this by listening for these changes and then triggering a save.

Summary

In this lesson we've gone a little bit beyond the beginner level and created a pretty robust data model. As I've mentioned, this certainly has it's benefits but don't feel too intimidated if you had trouble following along with this lesson - as a beginner you can mostly get away with just defining data directly on the class and not worrying about data models and observables.

I particularly don't want to freak you out too much with the Observabes - they are confusing (until you get your head around them) and outside of subscribing to responses from the Http service, you really don't have to use them in most simple applications. But once you do understand them, you can do some powerful stuff with them.

Although this lesson was a little more advanced, it's a great way to demonstrate how you might make

use of Observables in your project, and if you've kept up through this lesson then hopefully the next ones should be a breeze!