JOSHUA MORONY

# BUILDING MOBILE APPS WITH IONIC 2

# 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.

## 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!

# Changelog

**Version 10 (this version) - updated for RC.1**

- Minor bug fixes and improvements (no major changes)
- Moved data loading inside of platform.ready() calls

**Version 9 - updated for RC.0**

- Please see the official changelog for changes

**Version 8 - updated for beta.11**

- Changed implementation of overlay components
- Updated forms to use `formGroup` and `formControlName`
- Reformatted entire book to a more versatile format
- Added a .mobi and .ePub version of the book
- Various typo and formatting fixes

**Version 7 - updated for beta.10**

- Changed code to use `<ion-header>` and `<ion-footer>`
- Removed references to `*navbar`
- Switched to importing PouchDB instead of requiring, added a note on installing typings for Camper-Chat
- Formatting improvements
- Other minor typo and bug fixes

**Version 6 - updated for beta.9**

**Version 5 - updated for beta.8**

- Replaced @Page and @App with @Component
- Implemented ionicBootstrap
- For more details on changes in beta.8 see the Ionic 2 Changelog

**Version 4 - updated for beta.7**

- Updated to use TypeScript instead of ES6
- Updated to use @ViewChild instead of getComponent
- Updated syntax to match latest Angular RC

**Version 3 - updated for beta.5 / beta.6**

- Removed NgZone as it is no longer required
- Removed es6-shim import from app.js files
- Added changelog
- Added table of contents
- Fixed various typos

**Version 2 - updated for beta.4**

**Version 1 - Initial release**

## 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:

```html
<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:

```html
<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).

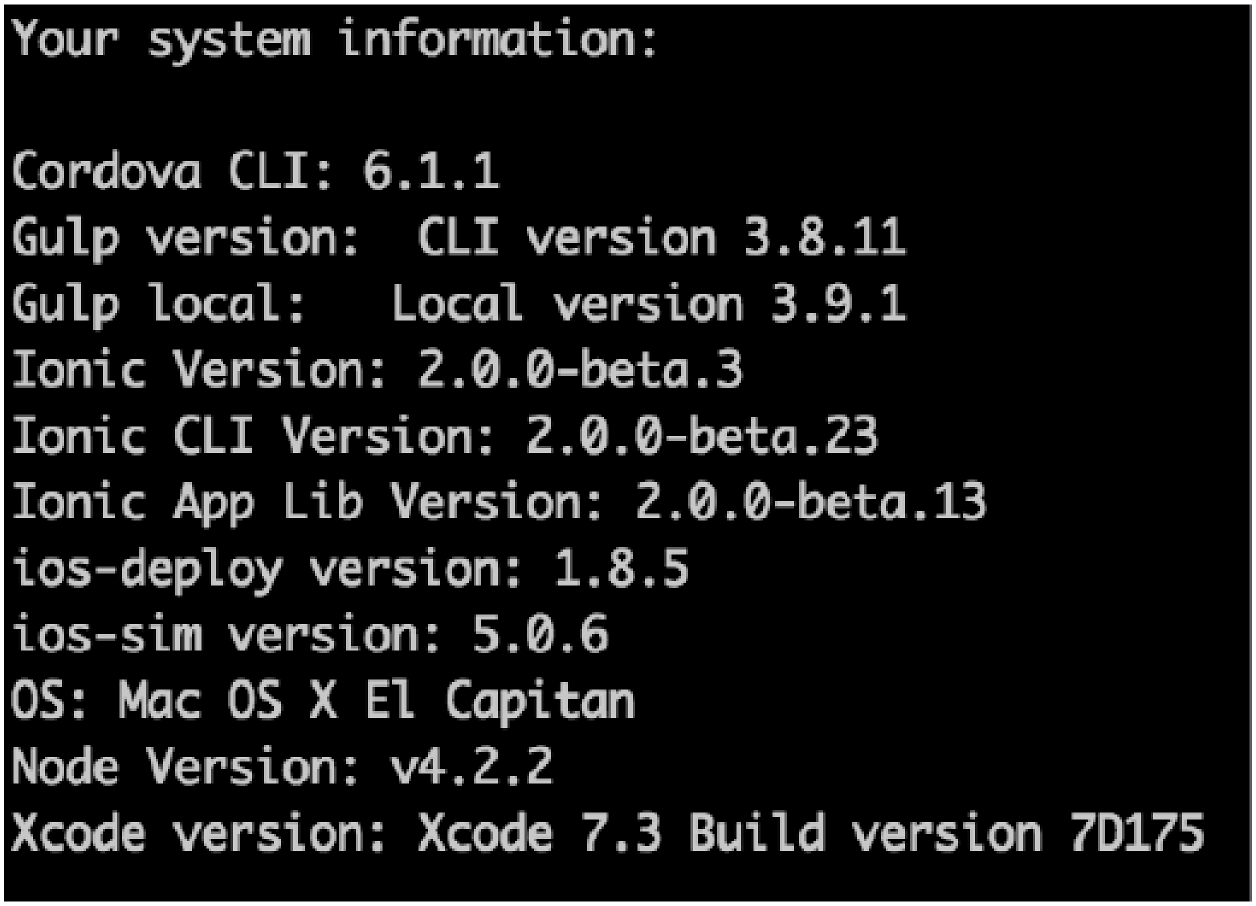
You should now have everything you need set up and ready to use on your machine! To verify that the Ionic CLI (Command Line Interface) is in fact installed on your computer, run the following command:

```
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:



```
Your system information:

Cordova CLI: 6.1.1
Gulp version:  CLI version 3.8.11
Gulp local:   Local version 3.9.1
Ionic Version: 2.0.0-beta.3
Ionic CLI Version: 2.0.0-beta.23
Ionic App Lib Version: 2.0.0-beta.13
ios-deploy version: 1.8.5
ios-sim version: 5.0.6
OS: Mac OS X El Capitan
Node Version: v4.2.2
Xcode version: Xcode 7.3 Build version 7D175
```

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

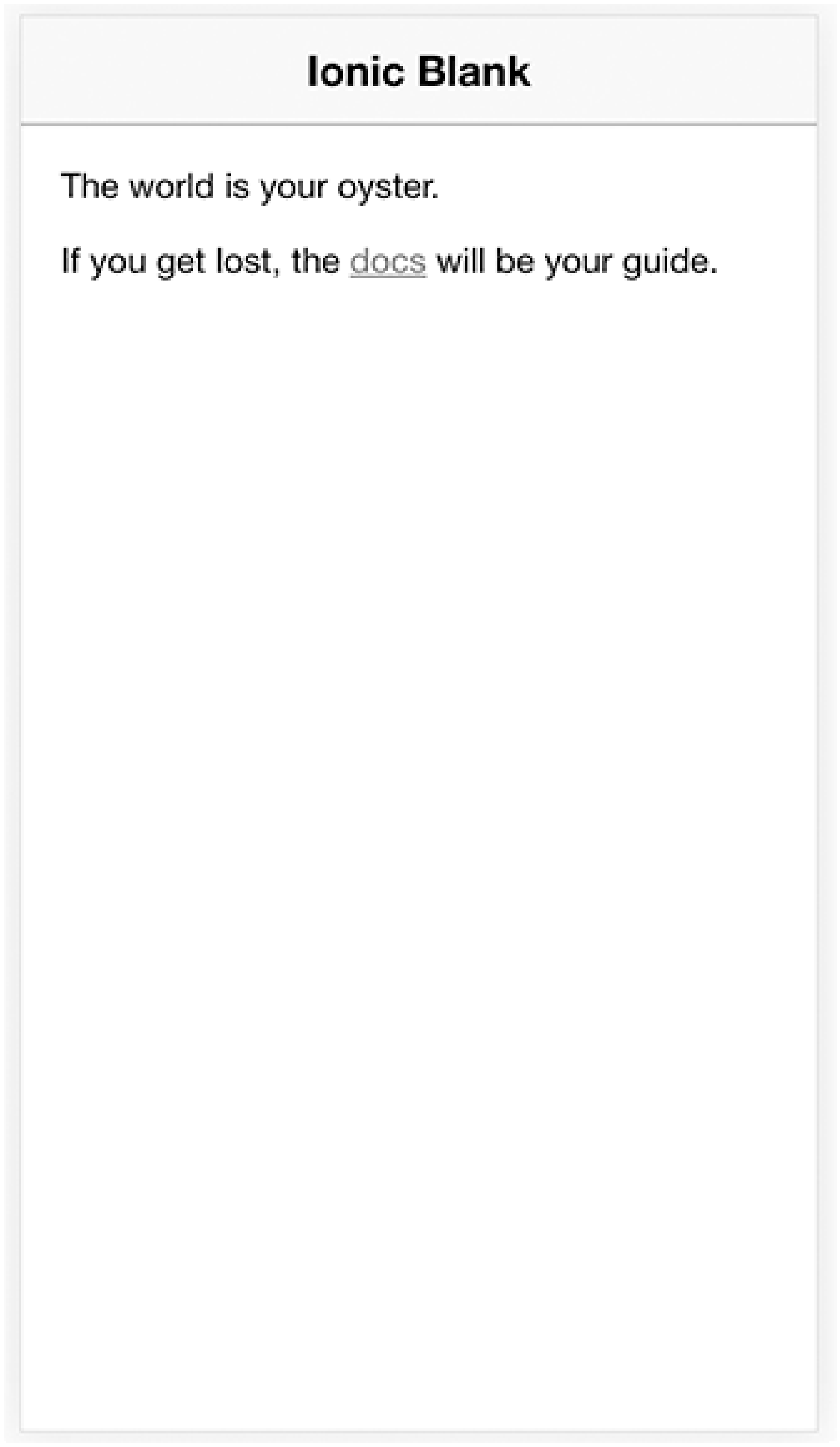
The beauty of HTML5 mobile applications is that you can run them right in your browser whilst you are developing them. But if you try just opening up your project in a browser by going to the **index.html** file location you won't have a very good time.

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:


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 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.

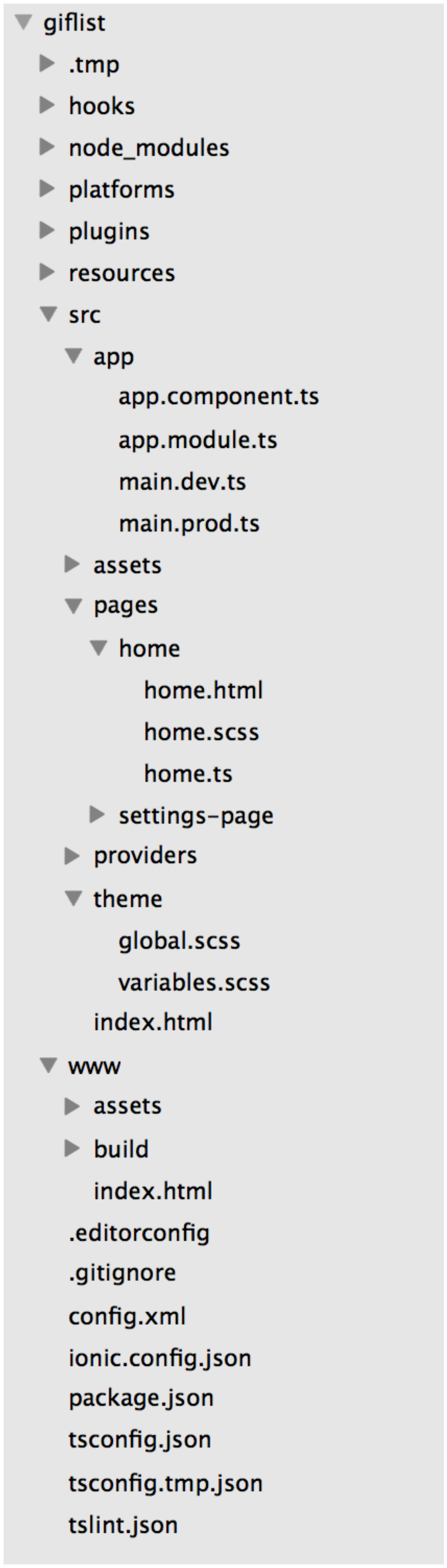
**IMPORTANT:** Keep in mind that there may be other dependencies in **package.json** that need to be updated, as well as the Ionic library.

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 2: Anatomy of An Ionic 2 Project

Now that we've covered how to get Ionic 2 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 2 application, your folder structure will look like this:



```
▼ giflist
  ▶ .tmp
  ▶ hooks
  ▶ node_modules
  ▶ platforms
  ▶ plugins
  ▶ resources
  ▼ src
    ▼ app
        app.component.ts
        app.module.ts
        main.dev.ts
        main.prod.ts
    ▶ assets
    ▼ pages
      ▼ home
          home.html
          home.scss
          home.ts
      ▶ settings-page
    ▶ providers
    ▼ theme
        global.scss
        variables.scss
      index.html
  ▼ www
    ▶ assets
    ▶ build
      index.html
    .editorconfig
    .gitignore
    config.xml
    ionic.config.json
    package.json
    tsconfig.json
    tsconfig.tmp.json
    tslint.json
```

At first glance, there's an intimidating amount of stuff there - but there's really not that much you need to worry about, and it will all make total sense with a little explanation. We're going to discuss what everything does, but I will cover the important stuff first in more detail (mainly the files and folders that you will be modifying) and then cover the less important stuff.

## Important Files & Folders

These files and folders are ones that you will be using on a frequent basis, so it's important that you understand their role. Fortunately, there's not too many of them!

**src**

This is where most of the action occurs. In a default application, your **src** folder will contain:

- An **app** folder
- A **pages** folder
- A **theme** folder
- An **assets** folder
- An **index.html** file

The **pages** folder will contain all of the page components for your application. If you look inside of the **pages** folder you should see another folder called **home**. This is a **component**, and is made up of a class definition (home.ts), a template (home.html), and style definitions (home.scss). For every page in your application you will add another folder here (we can actually get the Ionic CLI to do this for us automatically), so as well as the home folder you may also have **login**, **intro**, **checkout**, **about** and many more.

The **theme** folder will contain all of the **.scss** files which define application wide styles for the application. This includes a shared variables file that contains variables you can override, and a global file that can store generic styles for the whole application. There is a whole section later in this book dedicated to theming Ionic 2 applications so we'll discuss this in more detail there.

The **app** folder contains the **root component** for your application, **app.component.ts**. Again, we discuss what the root component is and what it does in detail in other parts of this book so I won't cover much right now, but essentially it's the "starting point" for your application. You will also find a **app.module.ts** which makes use of Angular 2's `@NgModule` which allows us to set up all of our application dependencies, like components and services we are using. Finally, you also have a `main.dev.ts` file and a `main.prod.ts` file, these are what kick off the bootstrapping process for your application. The `dev` file will be used in development, and the `prod` file will be used in production.

Bootstrapping just basically means setting up the **root component** of your application, which is essentially the first component that is created that will then go on and create all of the rest.

The **assets** folder is where you can store any static assets your application may want to make use of, like images and JSON files. These will be copied over to the build folder (so it's important you place assets here, **not** in the `www/assets` folder) when your application is built. You could for example create an images folder in here, and then reference those images in your application through `assets/images/myImage.png`.

Although you won't often have to, you can also edit the **index.html** file for your application ni the **src** folder, and like the **assets** this will also be copied over to the build directory.

As well as the default folders that your app will contain, as you start building your application you will likely see a few more folders in here as well, but we'll get to that later.

**IMPORTANT:** This is a really important concept to understand so make sure you read this over a couple of times if you don't get it initially. In the introduction section of this book we talked about webpack, transpiling and a bunch of other fancy ES6 stuff - the important thing to remember is that the ES6 and TypeScript features we are using aren't actually supported by browsers yet, so we need to "transpile" them into valid ES5 code. When you run or build your application, Ionic will bundle up everything in this app folder, perform all the magic it needs to perform, and then spits it out into the **www** folder.

When you are viewing your application through the browser you are actually viewing the bundled version inside of the **www** folder, not the code you created in the **app** folder. Likewise, when you build your application for release on iOS and Android (which we will discuss later) it is this **www** folder that will be used, not the **app** folder. **DO NOT EDIT CODE IN THE WWW FOLDER** - any changes you make in there (which is a bad idea anyway) will be overwritten when your app is rebuilt.

Using Angular 2 and the new ES6 syntax means the project structure and build process is quite a bit more complicated, but fortunately for us Ionic handles basically all of it for us. So don't worry too much about it, this is just something important to keep in mind so that you know what is going on.

#### config.xml

You won't need to edit this file very often, but it is a very important file. The **config.xml** file is basically used as a way to tell Cordova how to build your application for iOS and Android. You'll have to supply some important configuration information in here which we will discuss later, but you can also define some preferences in here (like whether the splash screen should auto hide, whether the app should be portrait only, and a bunch more). You will usually only really worry about this file when you're starting to look at getting your app on real devices.

## The Less Important Stuff

Obviously everything in your Ionic project plays a role, otherwise it wouldn't be there. But some of it is for more advanced use cases you might not need to worry about, and some of it is just pure configuration stuff that you'll never have to touch.

#### Configuration Files

If you take a look in the root folder of your generated project you will see a bunch of configuration files.

Aside from the **config.xml** file which we discussed above, you can pretty safely just ignore all of these. The only file you may want to edit is the **package.json** file to update the version of Ionic you are using.

**resources**

This folder simply contains the splash screens and icons that will be used for iOS and Android when you build your application. I'll show you a really easy way to create these resources with the Ionic CLI later.

**hooks**

Hooks are used as part of your applications build process, and you can add custom scripts in here to hook into various parts of the build process. At a beginner level you likely will not need to touch this at all, but if you wanted to start developing more complicated (but useful) workflows, related to versioning or deployment of your application for example, you could create some "hooks" here.

**node_modules**

This is another folder that you won't have to touch at all, but it's where all the goodies are stored. If you take a look in this folder you will find things like **ionic-angular**, **angular2**, and **ionicons**. This is where all of the source files for the various libraries that your application depends on are stored (including the Ionic framework itself).

## 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's 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.

I'm going to list out a few commands now and what they do, and since some commands have multiple different arguments that you can supply to them I will be using the following syntax:

```
ionic command [option1|option2]
```

in place of:

```
ionic command option1
```

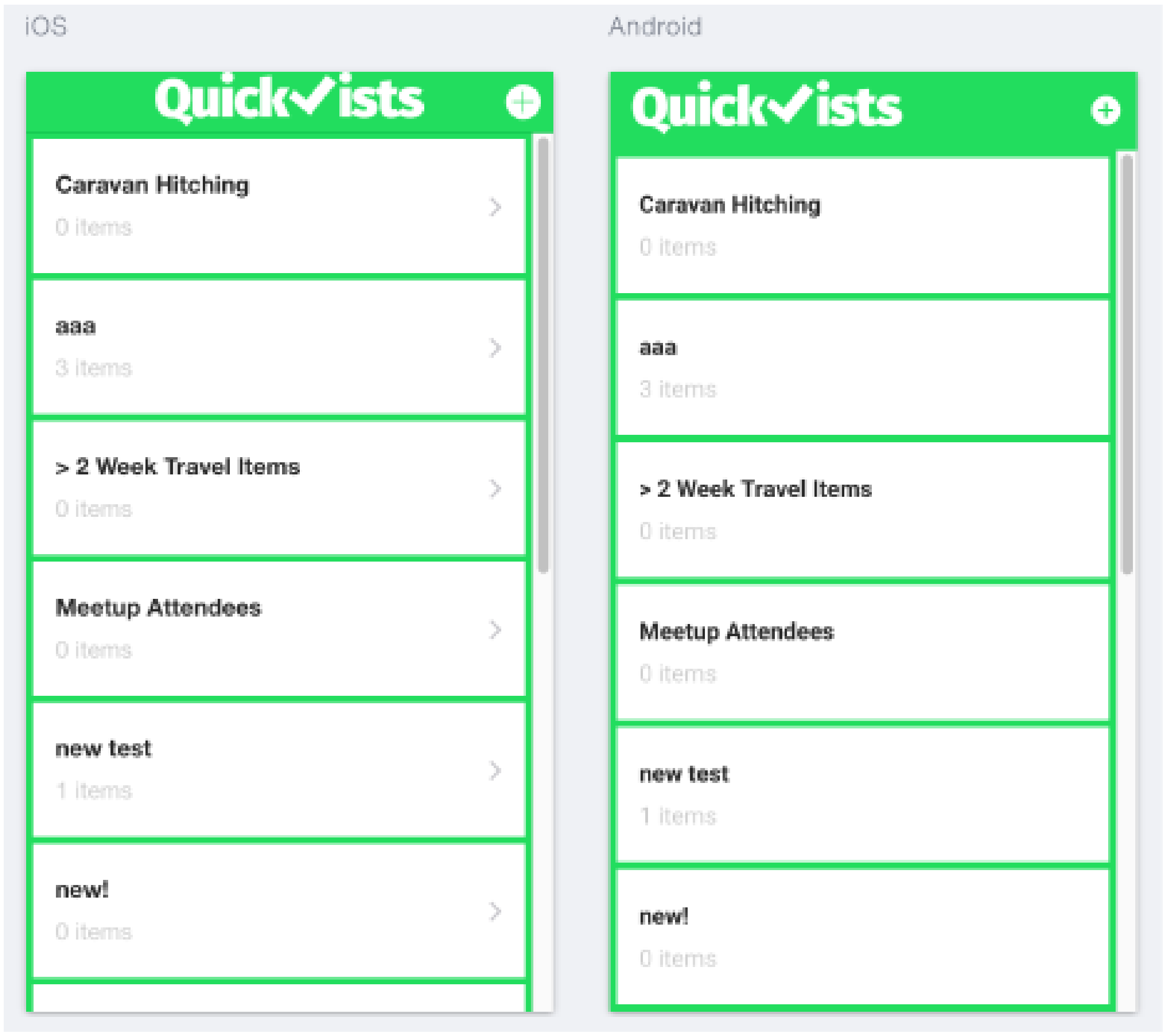
and

```
ionic command option2
```

for the sake of keeping things neat and tidy. Let's get into it!

```
ionic serve -l
```

You've already seen `ionic serve` which will launch your application in the browser with live reloading, but this command will launch Ionic Lab, which looks like this:



It will allow you to quickly see side by side what your application looks like on both iOS and Android.

```
ionic platform add [ios|android]
```

This will allow you to add platforms that you plan on building for.

```
ionic plugin add [plugin]
```

This will allow you to add Cordova plugins to your project, simply supply the plugin name.

```
ionic build [ios|android]
```

When you are ready to build your application for iOS and Android you can simply run these commands to build your project. There's a little bit more to it than this to actually get it on your device and then submit it to App Stores, but we will discuss all that later.

```
ionic run [ios|android]
```

If you want to test your application on a real device you can use this command to deploy the app directly to your device. For more information on this, please see the 'Testing & Debugging' section later in the book.

```
ionic emulate [ios|android]
```

Instead of deploying your application to a real device, this will launch an emulator on your machine and run it there instead.

```
ionic g [page|component|directive|pipe|provider|tabs]
```

This is the generate command and it is one of my favourites, it's a real time saver. As I mentioned before, your Ionic project will contain a bunch of components like the **home** page. 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. As well as pages, this command can also generate generic components, directives, pipes, providers and tabs. I'd recommend using it for any new component you are adding to your application.

```
ionic plugin list
```

This will allow you to see a list of all the plugins you have installed in your project.

```
ionic plugin rm [plugin]
```

This will allow you to remove any plugin from your project by supplying the plugin name.

```
ionic platform rm [android|ios]
```

This will allow you to remove a platform that you have previously added.

As I mentioned, this is not an exhaustive list but it does cover the most commonly used commands. I rarely use any commands outside of these.

## 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 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. 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.

# 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, what the class needs, 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 > 50;
    }
}
```

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** 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**, 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 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 your hotel room your scope is the room, if you leave your room your scope is now the hotel itself, if you leave the hotel your scope is now the world (or the country you are in if you prefer).

If you're not familiar with the this keyword, I'd recommend reading this.

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

```
var 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 50 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:

```
var john = new Person('John', 32);
var louise = new Person('Louise', 28);
var david = new Person('David', 52);

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 2

We know what a class is, but why are they suddenly a thing in Ionic 2 and Angular 2? 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 classes 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 > 50;
};

var david = new Person('david', 52);
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 2:

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

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

  constructor(public nav: NavController) {


  }


}
```

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 **NavController** from the Ionic library which we can use to control navigation.

We are also importing **SomePage** which is a class 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.

Next up we have the decorator, which we use to define the selector (i.e. the name this component will have in our DOM, `<home-page></home-page>`) 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 when the class is instantiated.

There's a little bit more to it that you need to know though. In Ionic 2 we need to inject any services that we want to use inside of this class into the constructor, which looks like this:

```
  constructor(platform: Platform, nav: NavController) {

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

    });

  }
```

In this example we want to use the **Platform** service to detect when the device is ready, so we inject it into the constructor and then we can use it within the constructor.

We don't need to use **platform** outside of the constructor here, but in most cases you will want to use the services you inject elsewhere in the class as well. So to make the service available to any function within the class, you must **set it as a member variable**. This will look like this:

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

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

  someMemberVariable: any = "hey"!;

  constructor(platform: Platform) {
    this.platform = platform
  }

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

    });
  }
}
```

**OR** we can use the `public` keyword I mentioned before when using TypeScript to automatically create these member variable references, like this:

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

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

  someMemberVariable: any = "hey"!;

  constructor(public platform: Platform) {

  }

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

    });
  }
}
```

Also note that any variables declared above the constructor like `someMemberVariable` is here will also automatically be set up as member variables. So we could access `someMemberVariable` from anywhere in this class through `this.someMemberVariable`. 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 **platform** from anywhere in this class through `this.platform`. If we did not set up this member variable and tried to access **platform** from our `someOtherMethod` function, it would not work.

## Creating a Page

Page's 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. Previously, there was a special decorator in Ionic for this, but now they are just regular **@Component**s.

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

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

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

  }
}
```

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

```
<ion-header>
 <ion-navbar>
  <ion-title>
    My Page
  </ion-title>
 </ion-navbar>
</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 } from '@angular/core';

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

  constructor() {
    //this runs immediately
  }

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

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

}
```

You could have your constructor call these other functions 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:** 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 a component), just as it will look like just about everything else we create. When creating pages with components we use Ionic's built in navigation to handle displaying them. A page is a component that will take up the whole screen (i.e. a "view" for the user), but a component will also allow you to create your own custom element that you can insert into your templates as well. Maybe you want to create a custom date picker component to add to your page, or a box that displays random motivational quotes - in both of these cases you would create a custom **Component**.

The class for a generic component will look like this:

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

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

  text: any;

  constructor() {
    this.text = 'Hello World';
  }
}
```

Really the only difference with the component here is that it specifies a **selector**. This will be the name of the element that you use to insert the component into your template, i.e:

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

Before we get to using it though, let's also talk 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.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 (as well as functions). 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. So let's take a quick look at how to now actually use the component in a page.

You will need to import it and add it to your **app.module.ts** file:

```
@NgModule({
  declarations: [
    MyApp,
    HomePage,
    MyComponent
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    HomePage
  ],
  providers: []
})
export class AppModule {}
```

and then you can just reference it in a pages template like this:

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

It's actually pretty unlikely that you will need to create your own custom components like this, 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 MyComponent`

## 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 2, on buttons for example:

```
<button ion-button>
```

or on your lists:

```
<ion-list no-lines>
```

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

```
<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 (it would be better to refer to it as a service or provider).

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(element: ElementRef) {
    this.element = element;
  }
}
```

You will also need to include this in your **app.module.ts** file just like the custom component.

**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 { Injectable, Pipe } from '@angular/core';

@Pipe({
  name: 'myPipe'
})
@Injectable()
export class MyPipe {
  transform(value, args) {

    //do something to 'value'

    return value;
  }
}
```

Notice that a pipe is also an **@Injectable**, we will talk about what that is in just a moment. So 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. Now 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 reference the pipe in your **app.module.ts** file before using it. "'

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

## Creating an Injectable

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’. The **@Injectable** that the Ionic CLI automatically generates looks like this:

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class Data {

  data: any;

  constructor(public http: Http) {
    console.log('Hello Data Provider');
  }

  load() {

    if (this.data) {
      return Promise.resolve(this.data);
    }

    // don't have the data yet
    return new Promise(resolve => {
      // We're using Angular Http provider to request the data,
      // then on the response it'll map the JSON data to a parsed JS object.
      // Next we process the data and resolve the promise with the new data.
      this.http.get('path/to/data.json')
        .map(res => res.json())
        .subscribe(data => {
          // we've got back the raw data, now generate the core schedule data
          // and save the data for later reference
          this.data = data;
          resolve(this.data);
        });
    });
  }

}
```

This code creates a provider called **Data** that loads data from a JSON source (which can either be a local JSON file, an external JSON file or a response from a server). It returns a promise, which will allow the data to be retrieved from the **http** request after it has finished executing. If the data has already been loaded then it just returns the data directly (also through a promise). We'll go into fetching data with **http** in more depth later, for now we just want to focus on the basics of an injectable.

So if we want to grab the data that this service returns, we would first inject it into wherever we want to use it, i.e:

```
import { Component } from '@angular/core';
import { Data } from '../../providers/data';

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

  constructor(public dataService: Data){

  }

}
```

add it the the `providers` array in **app.module.ts**:

```
providers: [Data]
```

and then you could make use of any function the injectable provides through `this.dataService`, for example:

```
    this.dataService.load().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. You might extend this provider further so that it also offered a **save** function, i.e:

```
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 provider MyProvider`

## Summary

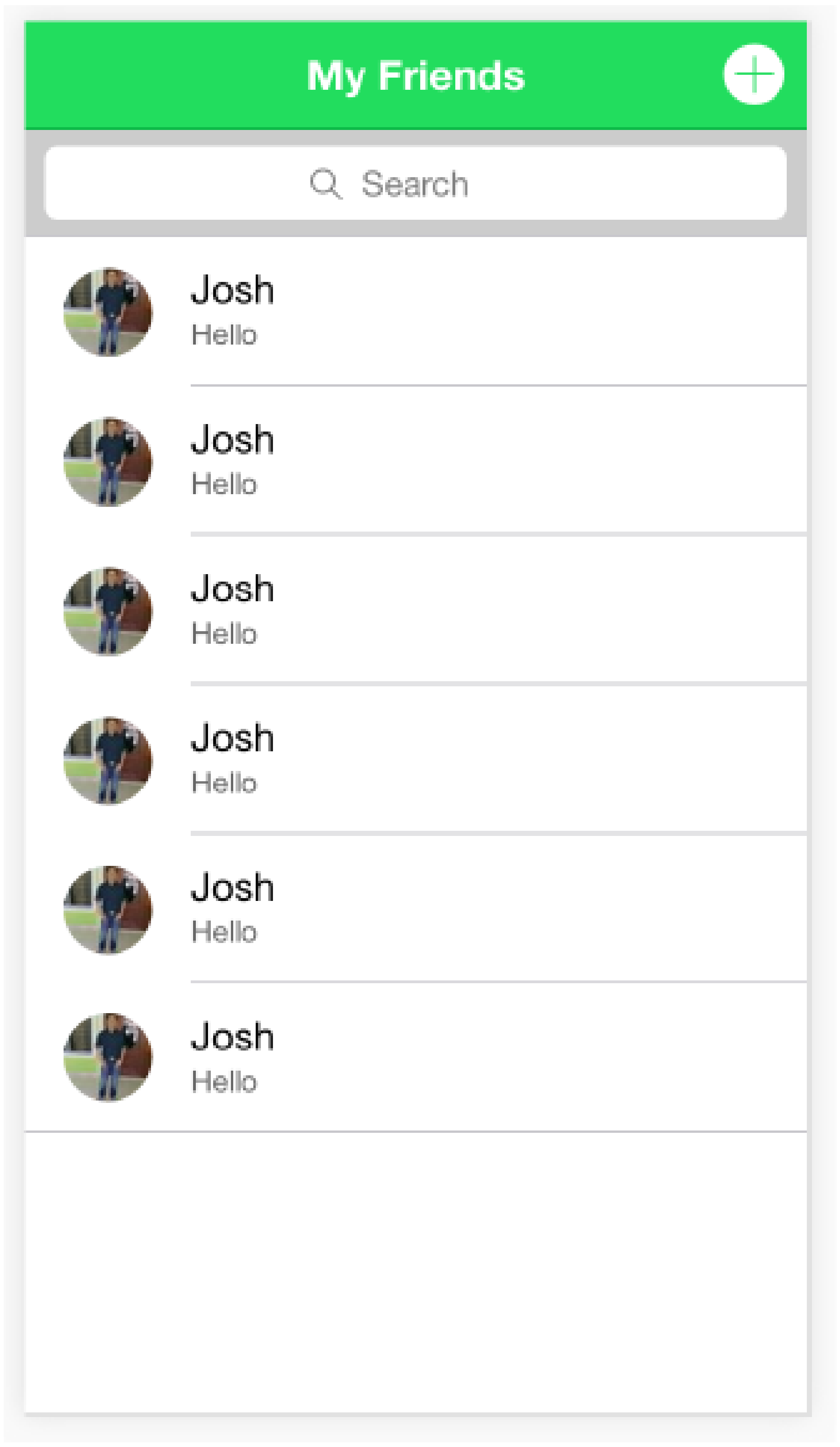
We've taken a pretty broad and basic look at how to create the different types of classes in Ionic 2, and there's certainly a lot more to know. But you should know enough now that it won't all look weird and foreign to you when we start diving into some real examples.

## Lesson 6: Templates

Templates, I think, are one of the most fun bits of Ionic 2. It's where the power of the framework really shines. Take this code for example:

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

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

<ion-content>
  <ion-searchbar (input)="getItems($event)"></ion-searchbar>
  <ion-list>
    <ion-item *ngFor="let item of items">
      <ion-avatar item-left>
        <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 2 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 2, and there's been a few significant changes from what you might have been used to in Ionic 1 - so we're going to dive into templates in a lot of detail.

We'll also be covering quite a few other topics after this, but this is the last of what I would consider to be the "core" knowledge required to get started with Ionic 2 - once you've got the basics of classes and templates down you can start jumping into building some stuff. So, strap in - we're going to start off with a little theory then get into some practical examples.

## The * Syntax

Perhaps one of the most confusing things about the new template syntax in Ionic 2 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 2 the `*` syntax is used as 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 using templates is that Angular 2 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 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:

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

So we need to use `<template>` to allow for this functionality, but writing out templates manually is a lot of work, and the `*` syntax just makes this a lot easier.

Now that we've got that out of the way, let's jump into some specifics of how to use 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 2 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.

## Conditionals

Sometimes you will want to display certain sections of the template only when certain conditions are met, and there's 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, it 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 *ngSwitchWhen="1">Paragraph 1</p>
  <p *ngSwitchWhen="2">Paragraph 2</p>
  <p *ngSwitchWhen="3">Paragraph 3</p>
  <p *ngSwitchDefault>Paragraph</p>
</div>
```

In this example we are checking the value of `paragraphNumber` with `ngSwitch`. Whichever `ngSwitchWhen` 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" item-left>
```

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 have a `this.hideAvatar` variable in your class definition which you could toggle to hide and show these elements.

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

```
<ion-avatar [class.my-class]="showMyClass" item-left>
```

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. 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 a different colour.

## Ionic 2 Template Components

Everything we've covered above is general Angular 2 stuff, there's nothing specific to Ionic there (except for the `<ion-list>` and `<ion-item>` elements we used). You will be using this syntax a lot throughout your templates, along with some Ionic specific components. We're going to go through some of the Ionic specific stuff now, starting with the basic layout of an Ionic 2 page template:

```
<ion-header>
 <ion-navbar>
  <ion-title>
    Home
  </ion-title>
 </ion-navbar>
</ion-header>


<ion-content class="home">
```

```
  <ion-card>
    <ion-card-header>
      Card Header
    </ion-card-header>
    <ion-card-content>
      Hello World
    </ion-card-content>
  </ion-card>
</ion-content>
```

This is the automatically generated code you will get for your template if you use the blank layout. There's two important components here that are used in just about every template and they are `<ion-navbar>` and `<ion-content>`.

The `<ion-content>` element is simply used to hold the main content of the page (which in this case is just a "card"), and allows for scrolling. Notice that it has a class of "home", if you look in the **home.scss** file you should also see a "home" class defined. This doesn't do anything special, it's just a convention to allow you to target styles to that `<ion-content>` specifically (remember, even though you add styles to the **home.scss** file they will still apply to the whole application, the separate file is purely for organisation).

The more interesting of the two is `<ion-navbar>`. This 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. It's not purely an aesthetic thing though, it also has a lot of inbuilt smarts for navigation. If you were to **push** a new page (a concept we will cover in detail later), then a back button will automatically be added to the `<ion-navbar>` which will automatically allow the user to navigate back to the previous page, rather than you having to handle that manually.

All of the above covers off on the basic template syntax you will see in a lot of your Ionic 2 pages, the rest is basically just dropping in and configuring various components that Ionic 2 offers (or if you're feeling adventurous, your own custom components).

Now let's take a look at how to implement a few of Ionic's components into our templates. We're not going to be covering anywhere near all of them because there is so many, I just want to give you a taste. For a full list of all the available components, take a look at the Ionic 2 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, and it all just *feels right* - well that's *really* hard to replicate. Fortunately, you don't have to worry about it because Ionic 2 does all the hard stuff for you, and using a list is as simple as adding the following to your template:

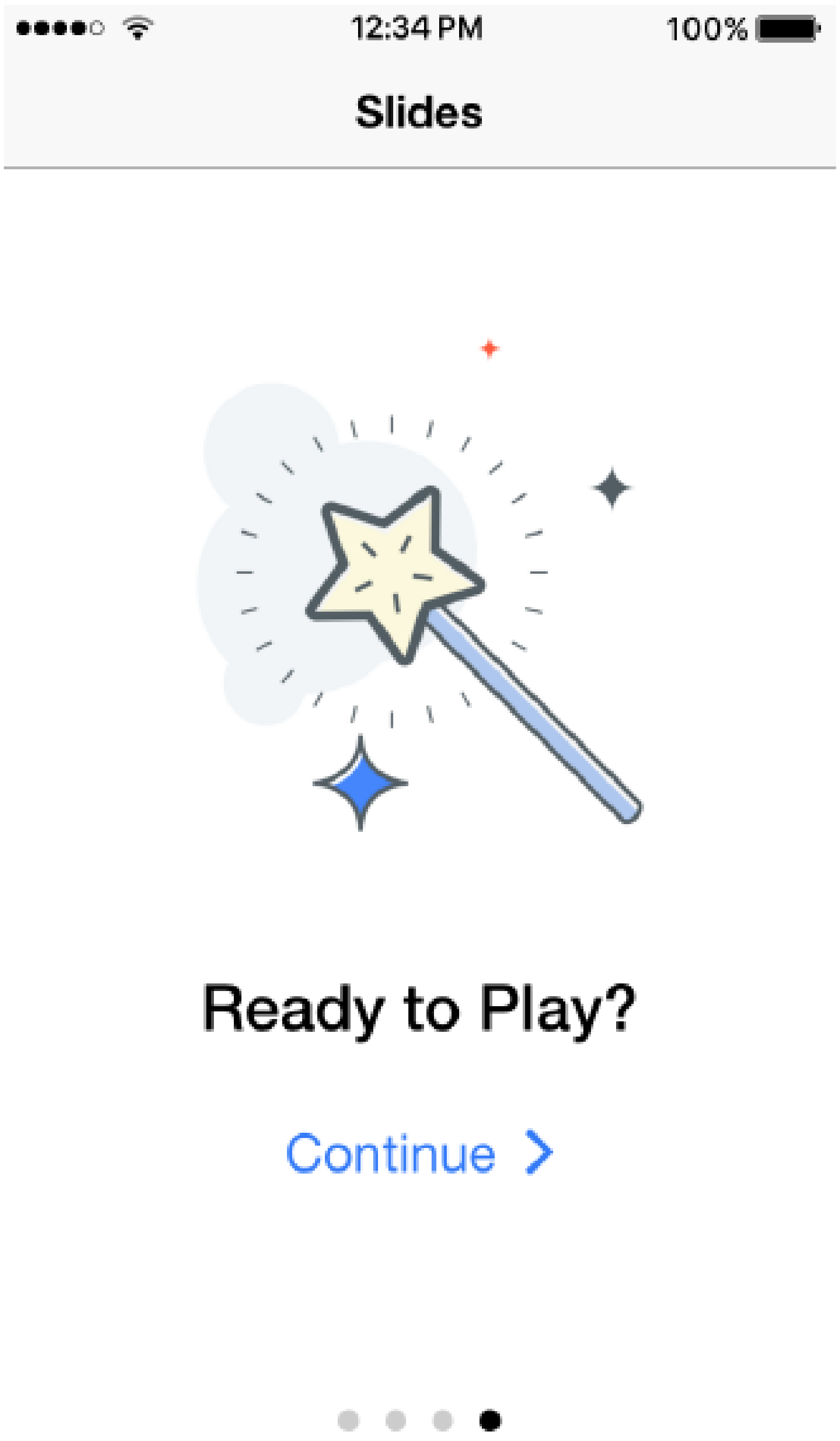
```
<ion-list>
    <ion-item>Item 1</ion-item>
    <ion-item>Item 2</ion-item>
    <ion-item>Item 3</ion-item>
</ion-list>
```

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)">
    {{item.title}}
  </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 2, creating slides is really easy as well:

```
<ion-slides [options]="slideOptions">

    <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; whether it should loop and whether it should have a pager for example (I'll give you a more complete example later).

**Input**

In Ionic 2, 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 fixed>Username</ion-label>
    <ion-input type="text" value=""></ion-input>
  </ion-item>

  <ion-item>
    <ion-label fixed>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 width-10></ion-col>
    <ion-col width-20></ion-col>
    <ion-col width-25></ion-col>
    <ion-col width-25></ion-col>
    <ion-col width-20></ion-col>
</ion-row>
```

This will create a single row with 5 different columns of varying widths (you will probably want to make sure your column widths add up to 100!).

For a full list of available widths 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 rather than relying on text. It's good for usability (most of the time) and looks way better than using a button that says something like "Add Item".

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

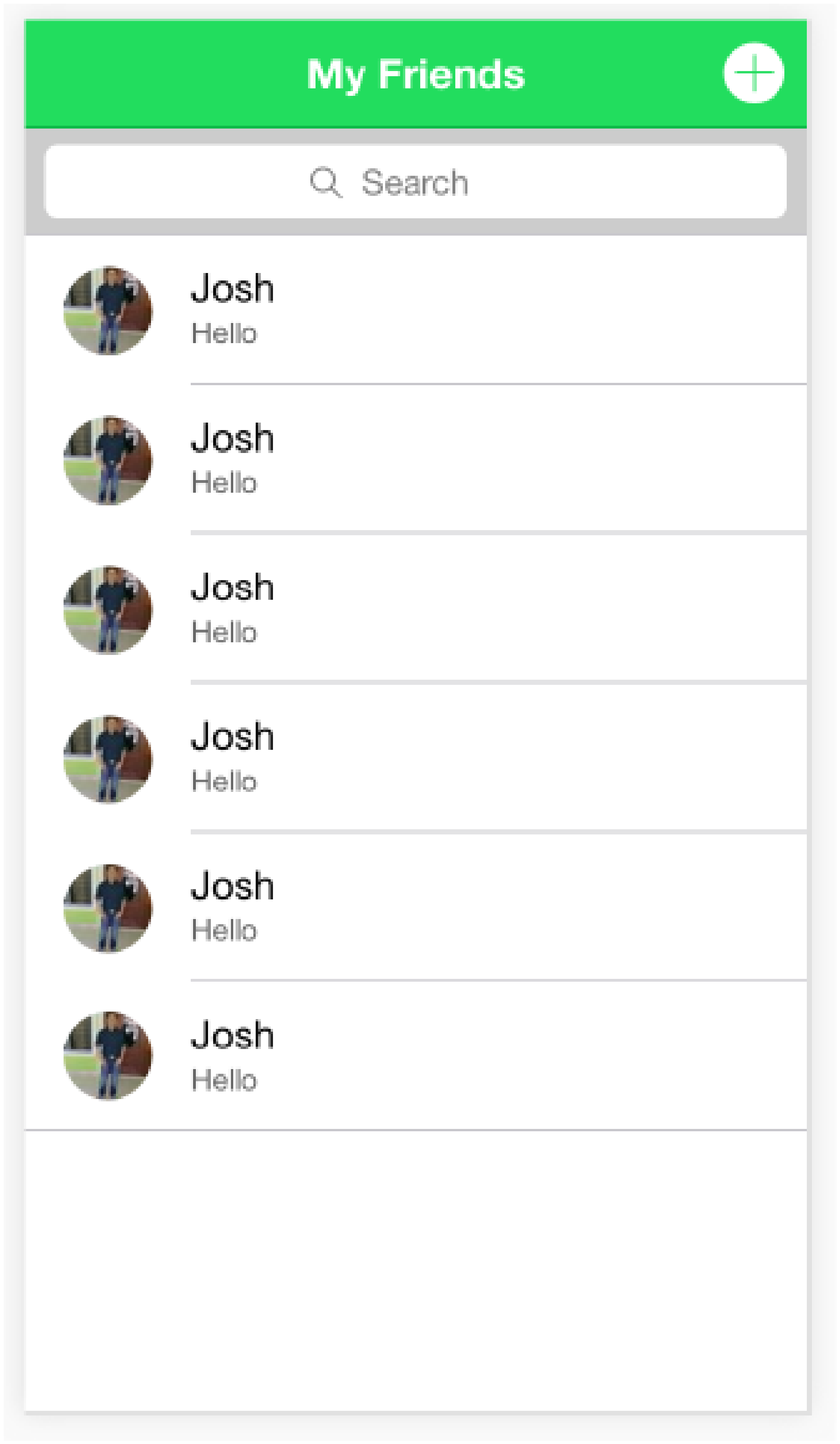
```
<ion-icon name="heart"></ion-icon>
```

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.

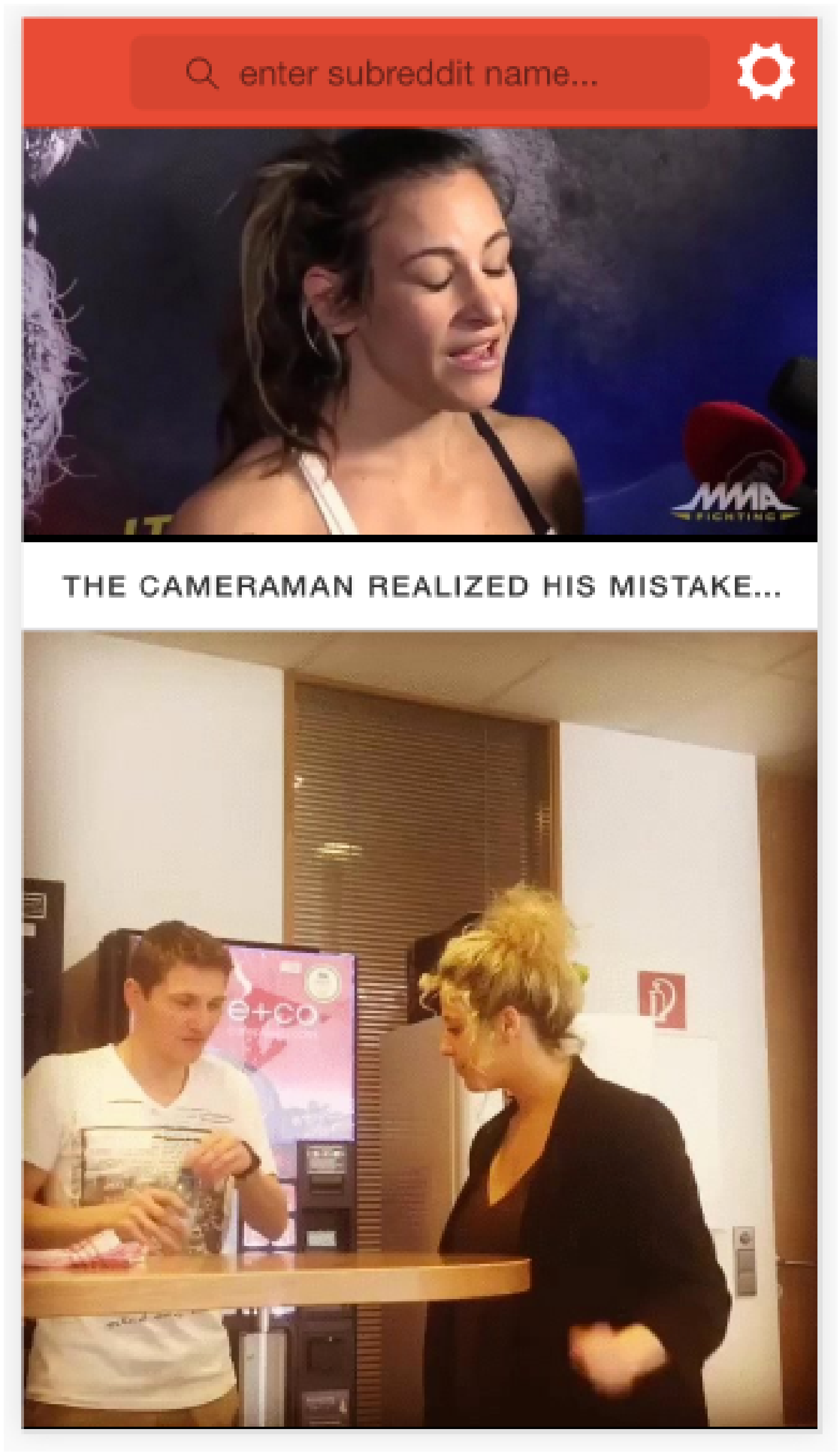
There's a ton more default components, and a lot more to know about even the ones I've mentioned here, so make sure you have a look through the documentation to familiarise yourself with what's available.

# Lesson 7: Styling & 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 course 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 2 applications, and the theory behind theming in general.

## Introduction to Theming in Ionic 2

When styling an Ionic 2 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**. Again, SASS isn't specific to Ionic or mobile web app development – it can also be used on any normal website – but many people may not be as familiar with SASS as they are with plain old 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 2, where we code using all the fancy new ES6 features, but that is then transpiled into ES5 which is actually supported by browsers now).

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.

If you take a look at the files generated when you create a new Ionic 2 project, you will see some **.scss** files inside of your **theme** folder, so let's quickly run through what their purpose is.

- **src/app/app.scss** is used to declare any styles that will be used globally throughout the application
- **theme/variables.scss** is used to modify the apps shared variables. Here you can edit the default values for things like `$colors` which sets up the default colours for the application, as well as `$list-background-color`, `$checkbox-ios-background-color-on` and so on. The general idea is that Ionic uses the variables defined in this file to determine styling for a lot of components, so it can be a great place to make quick changes. For a list of all the variables that you can overwrite, take a look at this page

On top of these **.scss** files inside of your **theme** folder, you will also have one for each component you create (or at least you should). To refresh your memory, most components you create in Ionic 2 will look like this:

**my-component**

- my-component.ts
- my-component.html
- my-component.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 create the `.scss` file for any components that have styling, rather than just defining the style in the **app.core.scss** file. If you use the auto generate commands the Ionic CLI provides then the .scss file will be created automatically for you anyway.

Why? Well you *could* just put all of your styles in the **app.core.scss** file and everything would work exactly the same, but there's 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 2 and Ionic 2 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 and dropped into other projects.

Now that we've gone over the theory, let's look at how to actually start styling our Ionic 2 applications.

## Methods for Theming an Ionic 2 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, SASS is used to define some colours, and these are:

- primary
- secondary
- danger
- light
- dark
- favorite

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

```
$colors: (
  primary:    #387ef5,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222,
  favorite:   #69BB7B
);
```

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. So if you add the **primary** attribute to most elements it will turn blue, or if you add the **danger** attribute it will be a red colour. But if you modified these then **primary** could make things purple and **danger** could make things pink.

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

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

or if I wanted to use the secondary colour on a the nav bar I could do this:

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

Keep in mind that these attributes aren't limited to just changing the colour of elements, some attributes will also change things like the position:

```
<ion-navbar color="secondary">
  <ion-buttons end>
    <button ion-button color="primary">I'm a primary coloured button in the
        end position of the nav bar</button>
  </ion-buttons>
</ion-navbar>
```

The example above uses the `end` attribute to decide where the buttons should appear. Also notice that we use the `ion-button` attribute with the button here, this let's Ionic know that we want to use the Ionic styling for the button. We could also control whether or not a list should have borders:

```
<ion-list no-lines></ion-list>
```

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

```
<ion-item detail-none></ion-item>
```

There's a bunch more of these attributes, so make sure to poke around the documentation when you are using Ionics in built 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 unneccesarily. This is why I recommend trying to do things with attributes first if you can, because you could save yourself a lot of effort.

**2. SASS Variables**

The next method you can use to control the style of your application is to change the default SASS variables (like editing the `$colors` we talked about above). These are really handy because it allows you to make app wide style changes to specific things. I touched on SASS variables before, but basically in your `.scss` files you can do something like this:

```
$my-variable: red;
```

and then you could reference `$my-variable` anywhere in the `.scss` file. So for example 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: $my-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:

- **$background-color**
- **$link-color**
- **$list-background-color**
- **$list-border-color**
- **$menu-width**
- **$segment-button-ios-activated-transition**

You can look at the documentation for more information on these and what they default to, but it's pretty clear by their name what they do. As you can see by the last example there, they even get very specific.

Editing these variables is really simple, just open **app.variable.scss** and insert your own definitions. Here's an example **app.variable.scss** from one of the applications in this book:

```
$colors: (
  primary:    #387ef5,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222,
  favorite:   #69BB7B
);

$list-background-color: #fff;
$list-ios-activated-background-color: #3aff74;
$list-md-activated-background-color: #3aff74;

$checkbox-ios-background-color-on: #32db64;
$checkbox-ios-icon-border-color-on: #fff;
```

```
$checkbox-md-icon-background-color-on: #32db64;
$checkbox-md-icon-background-color-off: #fff;
$checkbox-md-icon-border-color-off: #cecece;
$checkbox-md-icon-border-color-on: #32db64;
```

In this example some of the default colours have been changed, and some overrides for specific styles on both iOS and Android are provided.

Notice the use of **md** here, this stands for material design and is used for Android. Ionic 2 seamlessly adapts to the conventions of the platform it is running on with little to no style changes required from you – for Android this means material design is used.

The great thing about editing these default SASS 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. Configuration**

Another convenient way to change the styling of your application is through the **Config** object that you can provide to **IonicModule** in **app.module.ts**.

In general this is used for setting app wide defaults like the placement of buttons and tabs, the style of icons to be used, transitions and so on. Usually it's best to leave these unaltered unless you have a specific reason for changing it, since Ionic will adapt to the conventions of the platform it is running on automatically - messing with the config could break this.

Sometimes you will want to force things to be a certain way though, and the **Config** can be a good way to do that. Here's an example of what it might look like:

```
IonicModule.forRoot(MyApp, {
    backButtonText: 'Go Back',
    iconMode: 'ios',
    modalEnter: 'modal-slide-in',
    modalLeave: 'modal-slide-out',
    tabbarPlacement: 'bottom',
    pageTransition: 'ios'
})
```

and if you wanted to force iOS to use Material Design you could set the mode using the Config options:

```
IonicModule.forRoot(MyApp, {
    mode: 'md'
})
```

Again, I'd stress against doing something like this unless you have a good reason. You might like and be used to material design if you're an Android user, but your users on iOS (and vice versa) will not have the same view as you. With that in mind, the Config also allows you to configure things specifically for specific platforms like this:

```
IonicModule.forRoot(MyApp, {
    tabbarPlacement: 'bottom',
    platforms: {
     ios: {
       tabbarPlacement: 'top',
     }
    }
})
```

For more information on the Config object, take a look at the documentation.

**4. Custom Styles**

Before we talked about using attributes to change the colours of elements. Given that you can override these attributes 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. You can either define these custom classes in **global.scss** 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's a few different ways you can change the styling of your Ionic 2 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 SASS variables, because it will make your life easier.

As I mentioned before, Ionic seamlessly adapts to the UI conventions of both iOS and Android, so the more "hacky" or "brute force" your solution for styling is, the greater chance you have of breaking this behaviour.

# Lesson 8: Navigation

If you come from an Ionic 1 or Angular 1 background, then you would be used to handling navigation through routing with URLs, states and so on. The focus in Ionic 2 though is using a navigation stack, which involves **pushing** views onto the **navigation stack** and **popping** them off. Before we get into the specifics of how to implement this style of navigation in Ionic 2, let's try to get a conceptual understanding of how it works first.

## Pushing and Popping

Imagine your **root page** is a piece of paper that has a picture of a cat on it, and you put that piece of paper on a table. It is the only piece of paper currently on the table and you are looking down on it from above. Since it is the only piece of paper on the table right now, of course you can see the picture of the cat:

Now let's say you want to look at a different piece of paper (i.e. go to a different page), to do that you can **push** it onto the stack of papers you have. Let's say this one is a picture of a dog, you take that piece of paper and place it over the top of the picture of the cat:

The cat is still there, but we can't see it anymore because it is behind the dog.  Let's take it even further and say that now you want to **push** another piece of paper, a cow, it would now look like this:

Both the cat and the dog are still there, but the cow is on top so that is what we see.  Now let's reverse things a bit. Since all of the pieces of paper are stacked in the order they were added we can easily cycle back through them by **popping**. If you want to go back to the picture of the dog you can **pop** the stack of papers, removing the piece of paper that is currently on top (the cow). If you want to go back to the picture of the cat you can **pop** the stack of papers once more to remove the piece of paper that is now on top (the dog). Now we're back to where we started.

I'm sure you can see how this style of navigation is convenient for maintaining history and it makes a lot of sense when navigating to child views, but it doesn't always make sense to **push** or **pop**. Sometimes you will want to go to another page without the ability to go directly back to the page that triggered the change (a login screen that leads to the main app for example, or even just different sections of an app available through a menu).

In this case, we could change the root page which, given our pieces of paper on the table analogy, is like disregarding the other stack of papers we have and just focusing on a new piece of paper on the table:

In the example above, I've set the cow page as the root page, so rather than being on top of the other pages, it's all by itself.

At first, it may be hard to understand whether you should set the **root page** to navigate to a different page or push the view. In general, if the view you want to switch to is a child of the current view, or if you want the ability to navigate back to the previous view from the new view, you should **push**. For example, if I was viewing a list of artists and tapped on one I would want to **push** the details page for that artist. If I was going through a multi-page form and clicked 'Next' to go to page 2 of the form, I would want to push that second page.

If the view you are switching to is not a child of the current view, or it is a different section of the application, then you should instead change the **root page**. For example, if you have a login screen that leads to the main application you should change the root page to be your main logged in view once the user has successfully authenticated. If you have a side menu with the options **Dashboard**, **Shop**, **About** and **Contact** you should set the **root page** to whichever of these the user selects.

Keep in mind that **the root page is different to the root component**, typically the root component (which is defined in **app.ts**) will declare what the root page is – the root page can be changed throughout the application, the root component can not.

## Basic Navigation in Ionic 2

Ok, we've gone through the theory so now we're going to get into a more practical Ionic 2 example and look at how to **push**, **pop**, set the **root page** and even how to **pass data between pages**.

An important part of all this is the **NavController** which is provided by Ionic. You will often see this imported in Ionic 2 applications:

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

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

```
export class HomePage {

  constructor(public nav: NavController) {

  }
}
```

We inject the **NavController** and a reference to it is created so that we can use it anywhere within the class. As you might have been able to guess, the NavController helps us control navigation - so let's take a look at how to do just that by **pushing** and **popping**.

To push a page, which will take a page and put it on top of the navigation stack (which sets it as the current page), you can do something like this:

```
this.nav.push(SecondPage);
```

This uses the reference to the **NavController** we created before, and all you need to supply to it is a reference to the page that you want to navigate to, which you will need to make sure you also import at the top of the file, like this:

```
import {SecondPage} from '../second-page/second-page';
```

as well as adding it to both the `entryComponents` and `declarations` arrays in **app.module.ts** and that's it, your app should switch to the new page whenever the push code is triggered. When you push a page, a 'Back' button will automatically be added to the nav bar (assuming you have one), so you often don't need to worry about using pop to navigate back to the previous page since the 'Back' button does this automatically for you.

There may be circumstances where you do want to manually pop a page off of the navigation stack though, in which case you can use this:

```
this.nav.pop();
```

Easy enough right? As I mentioned before there is still another way to change the page and that is by setting the root page. If you take a look at your **app.ts** file you will notice the following line:

```
rootPage: any = MyPage;
```

Declaring `rootPage` in the root component will set the root page, and that's because the template for the root component looks like this:

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

So we're setting the `root` property on `<ion-nav>` to be whatever `rootPage` is defined as. To change the root page at any point throughout the application, you can use our friend the **NavController** – all you have to do is call the `setRoot` function like this:

```
this.nav.setRoot(SecondPage);
```

## Passing Data Between Pages

A common requirement of mobile applications is to be able to pass data between pages. One really common example is when using the "Master Detail" pattern, which is basically where you have a list of items and then you click on one to go to another page where it displays more details about that item. When navigating to the detail page, we're going to need to know which item we are displaying data for, which will involve passing in data from the previous page. In Ionic 2 this can be done using **NavParams**. First, you must pass through the data you want within the push call (this can also be done when using setRoot):

```
this.nav.push(SecondPage, {
    thing1: data1,
    thing2: data2
});
```

This is exactly the same as what we were doing before, except now there is an extra parameter which is an object that contains the data we want to send through to **SecondPage**. Then on the receiving page we need to import **NavParams** and inject it into our constructor:

```
import { Component } from '@angular/core';
import { NavController, NavParams } from 'ionic-angular';

@Component({
  selector: 'second-page',
  templateUrl: 'second-page.html'
})
export class SecondPage {

    constructor(nav: NavController, navParams: NavParams){

    }

}
```

Then you can grab the data that was passed through by doing the following:

```
this.navParams.get('thing1');
```

## Navigation Components

Some of the components that Ionic provides also effect navigation in some way. These aren't really core navigation concepts, but they will have an impact on navigation in your application. So let's cover what these are and when you might want to use them.

### Modals

You're probably familiar with the concept of a modal already. In web development, a modal is basically some box that pops up on the screen and covers the content behind it. Usually modals have the "lightbox" style, with a blacked out background and the focus on the content area.

A Modal in Ionic is similar, in that it pops up on top of your content, but it doesn't actually look any different to a normal page. Generally you would want to use a modal, rather than pushing a page, when you want to give the user the ability to launch and then dismiss (close) a view, rather than navigating back to the previous page.

One cool thing about Modals are that they give you the ability to pass some data back to the page that launched it when the Modal is dismissed. For example, you can create a Modal like this (remember to import and inject **ModalController** as well!):

```
let myModal = modalCtrl.create(MyPage);

myModal.present();
```

Notice that the modal is "presented", rather than being pushed onto the navigation stack. Now if we wanted to allow some data to be passed back from that modal, we could add an `onDismiss` handler to it before presenting it:

```
let myModal = modalCtrl.create(MyPage);

myModal.onDismiss(data => {
    console.log(data);
});

myModal.present();
```

Now when the modal is dismissed it will pass back a `data` object that we can do something with. To dismiss a modal, all you have to do is call the following code inside of the Modal:

```
this.view.dismiss();
```

where `this.view` is a reference to the **ViewController** which is kind of like the **NavController** and also needs to be imported and injected into your constructor. If we want to pass data back to that onDismiss handler though, we will need to do something like this:

```
let data = {
  thing1: "value1",
  thing2: "value2"
};

this.view.dismiss(data);
```

Now the `data` object will be passed back to the `onDismiss` handler from the page that launched the modal.

## Tabs

Tabs are a very popular component that have a big impact on how navigation works in your application. Using tabs is really simple, basically in your template you will create something like this:

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Tab 1" tabIcon="navigate"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="Tab 2" tabIcon="person"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Tab 3" tabIcon="bookmarks"></ion-tab>
</ion-tabs>
```

and then in your class definition you just define the pages to be used as the tabs like this:

```
tab1Root: any = TabOne;
tab2Root: any = TabTwo;
tab3Root: any = TabThree;

constructor(){

}
```

Notice that each tab has its own **root page**. You can think of switching tabs as switching between different root pages, and then you can push and pop pages in each tab. With a tab layout, you can switch between different tabs, but each tab will still maintain its own history.

**Sidemenu**

A side menu doesn't really do anything out of the ordinary in terms of navigation, the side menu is really just a UI element but it's a convenient, and common, place to add buttons that allow the user to navigate to another page (the actual switching of pages is just done manually with **setRoot** or **push** though). Adding a side menu to your application is super easy, you just need to modify the template of your root component to include it like this:

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

  <ion-content>
    <ion-list>
      <button ion-item (click)="openPage(homePage)">
        Home
      </button>
  </ion-content>
</ion-menu>

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

We use `<ion-menu>` to create a menu, and we also have to tell it what to attach itself to. This is why we set the [content] property to content which is a reference to the local variable we created on the `<ion-nav>` by adding `#content`. So this is basically saying that the `<ion-nav>` is our main content area, and we want the menu to attach to that.

There is a bit to learn about navigation in Ionic 2, but once you've got a handle on the basics discussed in this lesson you should be able to get by in most circumstances without too much trouble.

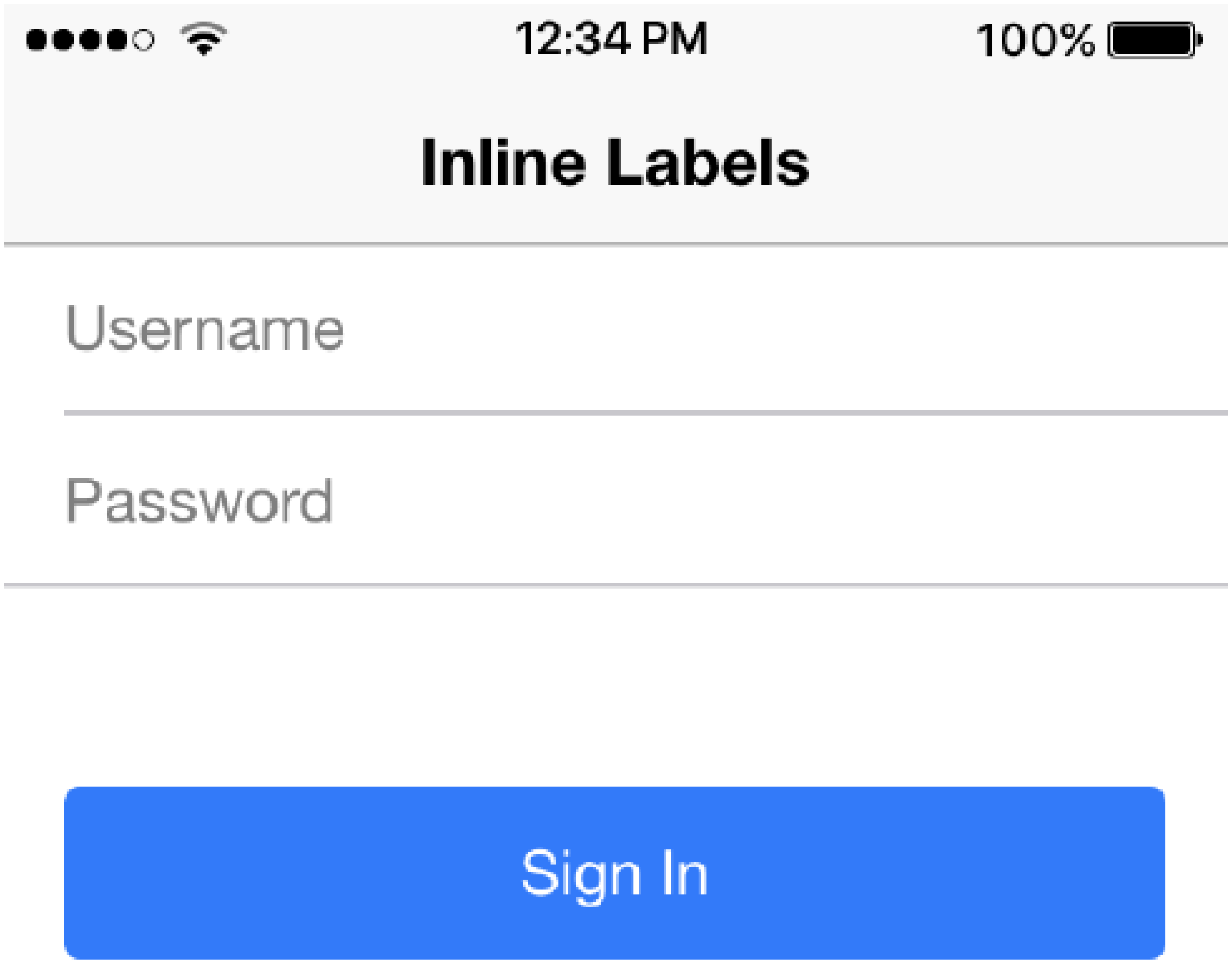
## Lesson 9: User Input

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 2 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-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

This concept will be very familiar to you if you've previously used Ionic 1, if not then don't worry because it's pretty straightforward concept. Two way data binding essentially links a value of an input field in the template, to the value of a 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>

<button ion-button (click)="logValue()">Log myValue!</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 screen. 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.

## Form Builder

Form Builder is a service provided by Angular 2, 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 { NavController } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  templateUrl: 'my-details.html',
})
export class MyDetailsPage {

  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" (submit)="saveForm($event)">

  <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>

  <button type="submit">Save Form</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 `(submit)` 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:

```
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(event){
    event.preventDefault();
    console.log(this.myForm.value);
}
```

We pass through the submit event and then call `preventDefault` so that the default action for submitting a form doesn't occur, we just want to handle it ourselves with this function. 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.

# Lesson 10: Saving Data

Let's say you've create an Ionic 2 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's 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 Cordova 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 2 applications, and they are **Local Storage** and **SQLite**.

## Local Storage

This is the most basic storage option available, which allows you to store up to **5MB** worth of data in the users browser. Remember, Ionic 2 applications technically run inside of an embedded browser.

Local storage gets a bit of a bad wrap, 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 can be easily accessed. 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 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.

## 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 a Cordova to easily gain access to this functionality. Simply run the following command in your project to install it:

```
ionic plugin add https://github.com/litehelpers/Cordova-sqlite-storage
```

The main benefits of using SQLite are that it:

- Provides persistent data storage
- There is no size limitation on how much can be stored
- It provides the SQL syntax, so it is a very powerful tool for managing data

Although there are some differences in supported commands between SQL and SQLite, it is almost exactly the same. Here's an example of how you might execute some simple queries in SQLite:

```
var db = window.sqlitePlugin.openDatabase({name: "my.db"});

db.transaction(function(tx) {
  tx.executeSql('DROP TABLE IF EXISTS test_table');
  tx.executeSql('CREATE TABLE IF NOT EXISTS test_table (id integer primary
      key, data text, data_num integer)');

  tx.executeSql("INSERT INTO test_table (data, data_num) VALUES (?,?)",
      ["test", 100], function(tx, res) {
    console.log("insertId: " + res.insertId + " -- probably 1");
    console.log("rowsAffected: " + res.rowsAffected + " -- should be 1");

  }, function(e) {
    console.log("ERROR: " + e.message);
  });
});
```

The example above looks pretty freaky, but if you're familiar with SQL then at least some of it should look familiar. This is the standard way to use SQLite with Cordova, but Ionic also provides its own service for using SQLite.

## Ionic Storage

Fortunately, for most storage requirements we don't really need to worry about the implementation details. Ionic provides it's own storage service that will automatically make use of the best available storage mechanism (whether that is plain old local storage, Web SQL, IndexedDB or SQLite). It provides us with a consistent API to use no matter which storage mechanism is being used underneath.

To use it, all you have to do it add it to your **app.module.ts** file by importing it:

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

and adding it to the provider array:

```
providers: [Storage]
```

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()
export class Data {

  constructor(public storage: Storage){

  }

  getData(): any {
    return this.storage.get('checklists');
  }


  save(data): void {
    let newData = JSON.stringify(saveData);
    this.storage.set('checklists', newData);
  }


}
```

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`.

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.

# 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 section 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 2 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:

```
['http://www.example.com/file1.jpg', 'http://www.example.com/file2.png',
   'http://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 'http://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 a 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 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.

## Observables and Promises

If you've used Ionic 1 or have a reasonably strong background in Javascript then you would probably be familiar with **Promises**, but far fewer people are familiar with **Observables**. Observables are one of the core new features included Angular 2 (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 a 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 into 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 than can emit more than one value over time**. It might be easier to think of Observables as streams, because they are, 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 a 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 situtations 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 indepth 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.

## 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. 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:

```
var 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 2 application?

The answer is to use the **Http** service which is provided by Angular 2, and allows you to make HTTP requests. If you're not familiar with what a HTTP request is, basically every time your browser tries to load anything (a document, image, a file etc.) it sends a HTTP request to do that. So we can make a 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 { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'page-one'
  template: 'page-one.html'
})
```

```
export class Page1 {

  constructor(public http: Http) {

  }
}
```

Since we have injected the Http service into our constructor and made it available through `this.http` by using `public`, we can now make use of it anywhere in this class. Also note that we are importing the map operator from the RxJS library. As I mentioned before map is a function that is provided by default on arrays - so why would we need to import it from some weird library? It's because the Http service doesn't return an array, it returns an **Observable**. The RxJS library makes the map function available for us on Observables, but we need to import it first.

Now let's take a look at how we might make a request to a reddit URL:

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'page-one'
  template: 'page-one.html'
})
export class Page1 {

  constructor(public http: Http) {

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


}
```

The first part of the call returns us an Observable. Then we make use of the **map** function, what we're doing here is taking the plain text JSON response (which is just a string) and converting it into a JavaScript object by calling the `json()`function. This makes it much more friendly for us to play with.

**IMPORTANT:** Remember, Http requests are asynchronous. This means that your code will continue executing whilst the data is being fetched, which would 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')
.map(res => res.json()).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')
.map(res => res.json()).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.

Getting back to our example, after we map the response we chain a subscribe call which allows us to do something with the data that is emitted from the stream (Observable). As I mentioned above 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:

```
this.subredditControl.valueChanges.debounceTime(1000)
.distinctUntilChanged().subscribe(subreddit => {
  this.subreddit = subreddit;

  if(this.subreddit != ''){
    this.changeSubreddit();
  }
});
```

In this case `subredditControl` is an **Observable**. 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 second (by using debounceTime) and it will also only run when a distinct value is supplied. I think this example shows well how a whole bunch of weird and useful stuff can be chained before the **subscribe** call to do a lot of useful things.

If you're looking at the example above and thinking "Wow… Ionic 2 is way too hard and confusing, take me back to Ionic 1!" then don't. This is a pretty advanced example using Observables to demonstrate a point, Ionic 2 input handling and two way data binding is just as easy as Ionic 1.

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 `http://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 `http://www.mywebsite.com/api/feed.php` in your `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.

This should give you a pretty reasonable overview of how to fetch remote data using Ionic 2 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

The problem with web based mobile applications is that whilst we can run them on iOS and Android through a web browser, users can't install them natively on their device and the app can not access native API's like Contacts, Bluetooth and so on. This is why we use **Cordova** in conjunction with Ionic, Cordova allows us to wrap our applications in a native wrapper which allows for submission to app stores as well as communication with native API's through plugins. When using Cordova, a HTML5 mobile application can do just about anything a normal native application can do.

As I just mentioned, to access native functionality we need to use plugins. Cordova provides a bunch of default **plugins** which include:

- Device
- Network Information
- Camera
- Geolocation
- File
- In App Browser
- Media
- Splash Screen

But there are hundreds of open sourced plugins developed by the community to do just about everything, a few popular examples being:

- Local Notifications
- Facebook Connect
- SQLite
- Social Sharing

Basically what a plugin does is create an interface where Javascript code can trigger native calls. So if you ever run into a situation where you need a Cordova plugin that doesn't exist yet (which is pretty rare), you can even write it yourself (but it does involve writing native code).

**IMPORTANT:** Most plugins only work when running on a real device, so if you are trying to test a Cordova plugin through ionic serve you will likely receive errors.

## Using Cordova Plugins in Ionic 2

There's two ways to implement native functionality in Ionic 2. You can just use any Cordova plugin directly by installing the plugin in your project:

```
ionic plugin add plugin-name
```

and then accessing the functionality that the plugin provides, which is usually available on a global object like this:

```
window.plugins.somePlugin.someMethod();
```

Nothing needs to be imported, required, called from a specific section of code or anything else - once you have installed the plugin through the command line you will be able to access it from anywhere. Not all plugins will be accessible exactly like this, but it's how most plugins work. This is not specific to Ionic 2, you can use Cordova plugins in this manner in *any* Cordova project (the only difference being that you would use `cordova plugin add` instead of `ionic plugin add`). When using the normal Cordova syntax, using a plugin in Ionic 2 is no different than using it in Ionic 1, Sencha Touch, jQuery Mobile or a normal web page built with Cordova.

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. 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.

Alternatively, you can use **Ionic Native** to make use of Cordova plugins, which is specific to Ionic 2. If you're familiar with **ngCordova** from Ionic 1 then this is basically the same thing, just for Ionic 2. If you're not familiar with ngCordova, Ionic Native basically just makes Cordova plugins play a little bit more nicely with Angular 2, by adding support for Promises and Observables.

Ionic Native is installed by default in all Ionic 2 applications, so all you need to do is install the plugin you want to use, just like you would normally, for example:

```
ionic plugin add cordova-plugin-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';
```

and then you can use it in your code:

```
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.

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).

Although it is not required, you should use Ionic Native wherever possible. It'll make your code cleaner, and it makes much more sense in the Angular 2 ecosystem (and typings will be handled automatically this way, so TypeScript won't complain). Using plain old Cordova is not a crime though, so don't feel too bad about it.

# Chapter 3

# Quick Lists

# Lesson 1: 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 because it covers a broad range of the core concepts in Ionic 2, 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 strucutre & 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.

## About Quick Lists

The idea for Quick Lists came from a personal need of mine. At the time of writing this I am currently 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's 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, and when you needed to restart the checklist for the next time you could just hit a refresh button to reset everything. A *repeatable* todo list application in a sense.

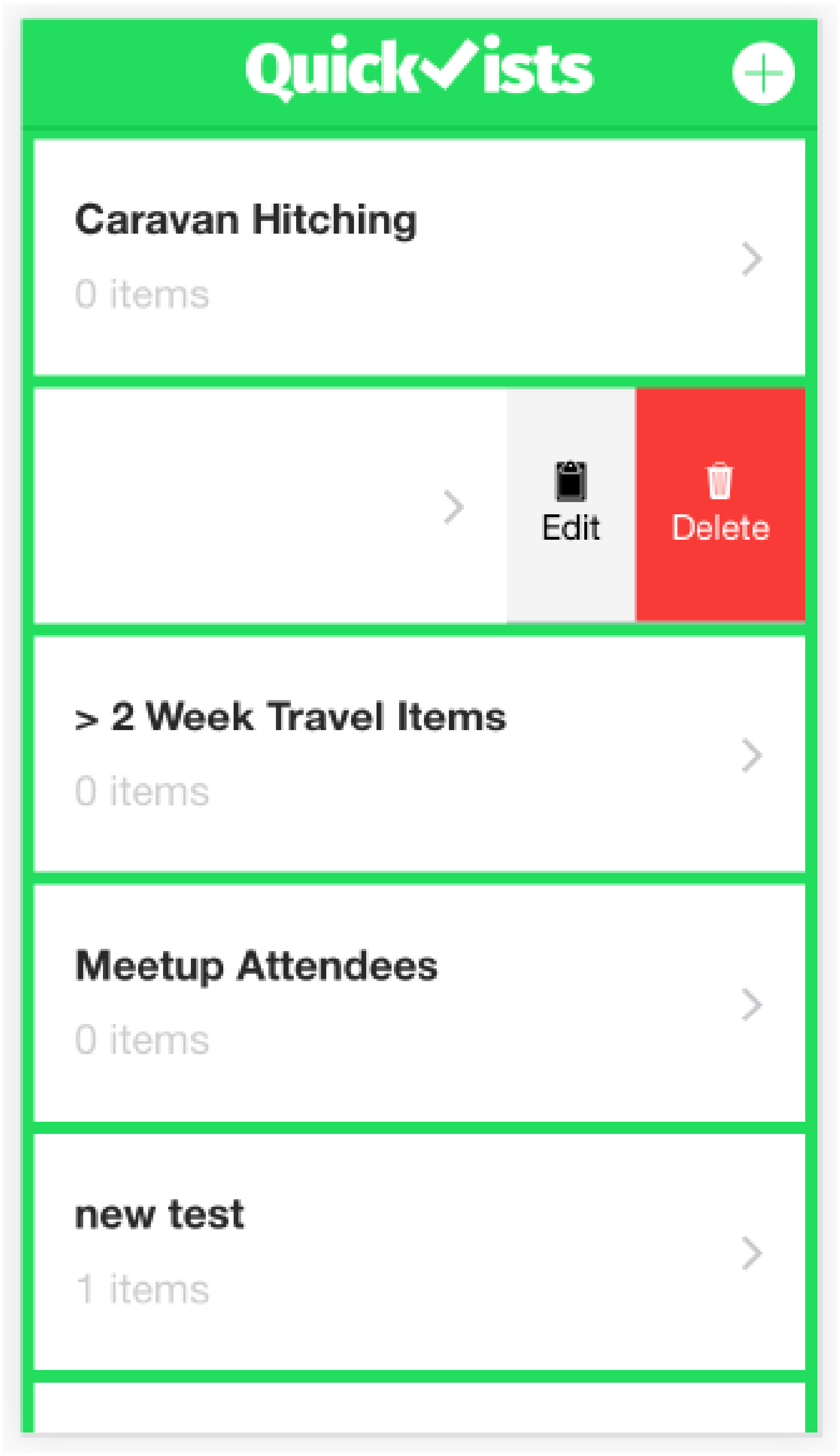
As I mentioned, this application covers similar concepts to what a traditional todo application would. Specifically though, the main concepts we will cover are:

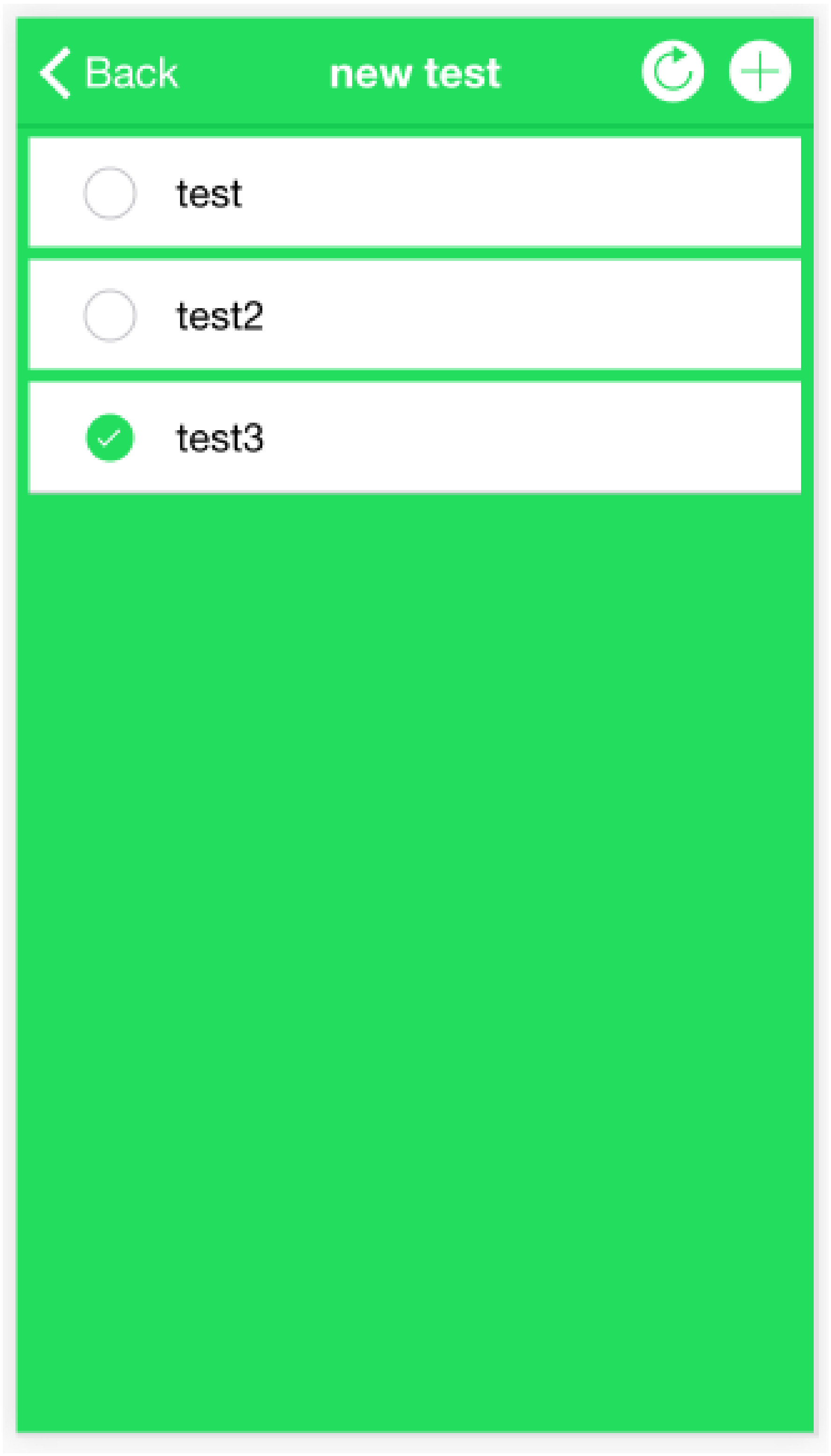
- Complex Lists
- Data Models
- Observables
- Forms and User Input
- Simple Navigation
- Passing Data Between Pages
- Creating, Reading, Updating and Deleting Data
- Data Storage and Retrieval
- Theming

Here’s a quick run down on 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 “reset” a checkist at any time
- 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.


QuickLists
Caravan Hitching
0 items
Edit
Delete
> 2 Week Travel Items
0 items
Meetup Attendees
0 items
new test
1 items



###Lesson Structure

**1. Getting Ready**

**2. Basic Layout**

**3. Data Models and Observables**

**4. Creating Checklists and Checklist Items**

**5. Saving and Loading Data**

**6. Creating an Introduction Slider & Themeing**

**Ready?**

Now that you know what you're in for, let's get to building it!

## Lesson 2: Getting Ready

In this lesson we are going to prepare our application for the journey ahead. We are going to of course generate the application, and we are also going to set up all of the components and Cordova plugins we will need. 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 Ionic and Cordova, so if you haven't done it recently then make sure to run:

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

or

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

before you continue. 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.

### 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 **home** 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 --v2
```

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

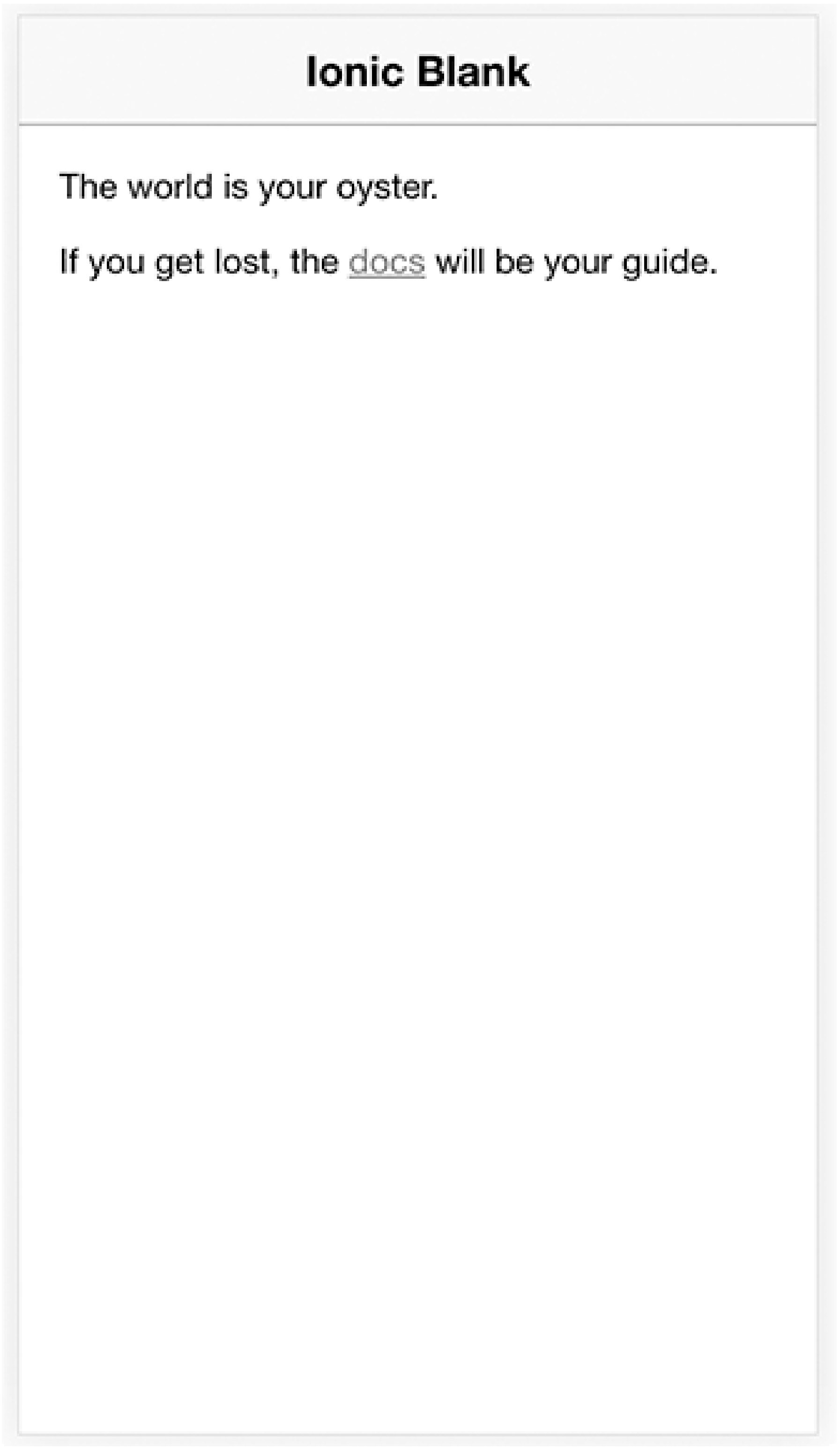
```
cd quicklists
```

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:



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.

**> Run the following command to generate the Introduction page:**

```
ionic g page IntroPage
```

**> Run the following command to generate the Checklist detail page:**

```
ionic g page ChecklistPage
```

## 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 Data provider:**

```
ionic g provider Data
```

## Create the Model

We will be creating a data model for our checklists which will allow us to more easily create and update them. Unforuntately, there is no generate command for this, so you will need to create it manually.

**> Create a new folder inside of the src folder called models**

**> Create a new file inside of the models folder called checklist-model.ts**

## Add Pages & Services to the App Module

In order to use these pages and services throughout our application, we need to add them to our **app.module.ts** file. All of the pages we created need to be added to both the `declarations` array and the `entryComponents` array, all of the providers we create need to be added to the `providers` array, and any custom components or pipes only need to be added to the `declarations` array. Our model is just a simple class we will import wherever we need, there is no need to set it up anywhere in the module.

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

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { IntroPage } from '../pages/intro-page/intro-page';
import { ChecklistPage } from '../pages/checklist-page/checklist-page';
import { Data } from '../providers/data';

@NgModule({
  declarations: [
    MyApp,
    HomePage,
    IntroPage,
    ChecklistPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    HomePage,
    IntroPage,
    ChecklistPage
  ],
  providers: [Storage, Data]
})
export class AppModule {}
```

Notice that we also have a `Storage` provider as well as the `Data provider we created.`Storage‘ is provided by Ionic and allows us to save and retrieve data - we will be making use of this later.

## Add Required Platforms

Before you can build for certain platforms, you need to add them to your project. This is something we will be doing way later on in this course, but you may as well just set them up now.

**> Run the following command to add the iOS platform to your application**

```
ionic platform add ios
```

**> Run the following command to add the Android platform to your application**

```
ionic platform add android
```

### Add Required Cordova Plugins

This application will use a few different Cordova plugins. Remember, Cordova plugins can only be used when running on a real device. I'll explain each plugin as we run through the commands for adding them.

**> Run the following command to add the SQLite plugin:**

```
ionic plugin add https://github.com/litehelpers/Cordova-sqlite-storage
```

This plugin gives you access to native storage with an SQLite database. We are adding it to this application because the Ionic local storage service can make use of this plugin to provide more stable data storage.

**> Run the following command to add the Status Bar plugin:**

```
ionic plugin add cordova-plugin-statusbar
```

We will be adding this plugin to all projects to give us control over the status bar in our application (the bar at the top of the devices screen that contains the time, battery information and so on).

**> Run the following command to add the Splash Screen plugin:**

```
ionic plugin add cordova-plugin-splashscreen
```

This plugin allows us to control the splash screen (the fullscreen graphic that briefly displays when you open an app)

**> Run the following command to add the Whitelist plugin:**

```
ionic plugin add cordova-plugin-whitelist
```

This plugin is required for all applications, and helps to define what resources should be allowed to be loaded in your application. Without it, resources you try to load will fail.

As well as adding the plugin, you also need to define a "Content Security Policy" in your **index.html** file. We will be adding a very permissive policy which will essentially allow us to load any resources. Depending on your application, you may look into providing a more strict policy, but an open policy is good for development.

**> Modify your `src/index.html` file to include the following meta tag:**

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
   img-src * data:; default-src * 'unsafe-eval' 'unsafe-inline'">
```

**> Run the following command to add the Crosswalk plugin:**

```
ionic plugin add cordova-plugin-crosswalk-webview
```

This is another plugin that we will add to every application, but you may decide you want to leave it out. By adding this plugin, when you build for Android "Crosswalk" will be used. Android has a lot of issues, especially with older devices, because there is so many different software versions out there and different versions have different browsers (remember, since we are building HTML5 applications it is actually a browser engine powering and running our application). What Crosswalk does is bundle a modern browser into your application so no matter what device you are running on your app will be powered by the same browser, and the Crosswalk browser can improve performance a lot.

The only real downside to this is that it increases the size of your application by a significant amount. In general, I think it's worth it and I'd recommend you include it but you may leave it out if you like. For more information take a look at the Crosswalk Project website: https://crosswalk-project.org/

## 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.

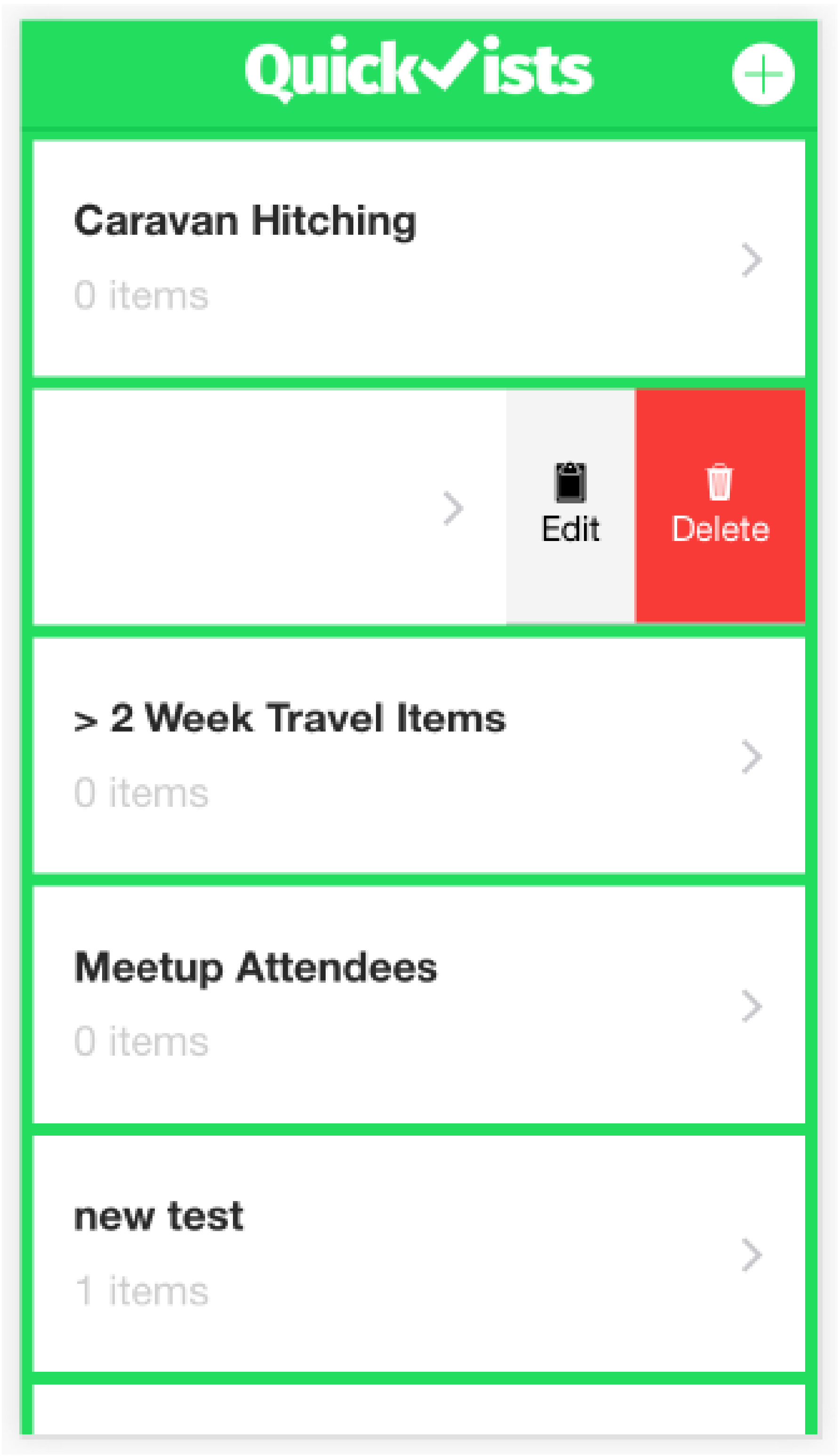
# Lesson 3: Basic Layout

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 home page:



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 component Ionic provides, which allows us to specify some content that will be displayed when the user swipes the list item.

So 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/pages/home/home.html` to reflect the following**

```
<ion-header>
    <ion-navbar color="secondary">
      <ion-title>
        <img src = "assets/images/logo.png" />
      </ion-title>
      <ion-buttons end>
        <button ion-button icon-only (click)="addChecklist()"><ion-icon
            name="add-circle"></ion-icon></button>
      </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
  <ion-list no-lines>

      <ion-item-sliding>

        <button ion-item (click)="viewChecklist(checklist)">
            TITLE GOES HERE
            <span>0 items</span>
        </button>

        <ion-item-options>
          <button ion-button icon-only color="light"
```

```
             (click)="renameChecklist(checklist)"><ion-icon
             name="clipboard"></ion-icon> Edit</button>
          <button ion-button icon-only color="danger"
             (click)="removeChecklist(checklist)"><ion-icon
             name="trash"></ion-icon> Delete</button>
        </ion-item-options>

      </ion-item-sliding>

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

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

```
<ion-header>
    <ion-navbar color="secondary">
      <ion-title>
        <img src = "assets/images/logo.png" />
      </ion-title>
      <ion-buttons end>
        <button ion-button icon-only (click)="addChecklist()"><ion-icon
           name="add-circle"></ion-icon></button>
      </ion-buttons>
    </ion-navbar>
</ion-header>
```

The `<ion-navbar>` allows us to add a header bar to the top of our application that can hold buttons, titles and even integrates directly with Ionic's navigation system to display a back button when necessary.

We add the color attribute to the navbar to style it with our secondary colour, which is defined in **theme/-variable.scss**. Inside of this navbar 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 navbar.

By specifying the end attribute, the buttons will be placed on the right side on iOS, but if we were to specify the 'start' attribute it would be displayed on the left instead. Remember that Ionic 2 has platform continuity built in, so by default the buttons will be placed in the most appropriate place for the platform they are running on.

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.ts** file (which we have not created yet of course). Also notice that we use the `ion-button` and `icon-only` attributes on the button, this lets Ionic know that we want to add the Ionic styling to this button, and that it should style it as a button with just an icon and no text.

Let's move onto the list section now:

```
<ion-content>
  <ion-list no-lines>


      <ion-item-sliding>


        <button ion-item (click)="viewChecklist(checklist)">
            TITLE GOES HERE
            <span>0 items</span>
        </button>


        <ion-item-options>
          <button ion-button icon-only color="light"
              (click)="renameChecklist(checklist)"><ion-icon
              name="clipboard"></ion-icon> Edit</button>
          <button ion-button icon-only color="danger"
              (click)="removeChecklist(checklist)"><ion-icon
              name="trash"></ion-icon> Delete</button>
        </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 navbar 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 `no-lines` attribute to the `<ion-list>`. Just like the `secondary` attribute we added to navbar, this attribute also styles our list except that it will cause the items in the list to not display with borders.

The next bit get's a little trickier, as it is where we set up our sliding item. An `<ion-sliding-item>`, 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-sliding-item>` is the normal `<ion-item>` definition, but instead of using `<ion-item>` directly we are using `<button ion-item>` which is actually a button with the styling of an item. Visually, these two methods are exactly the same, but on mobile everything that is not a `<button>` or `<a>` element that has a click handler will have a slight tap delay. We don't want to have this delay so we use the button instead.

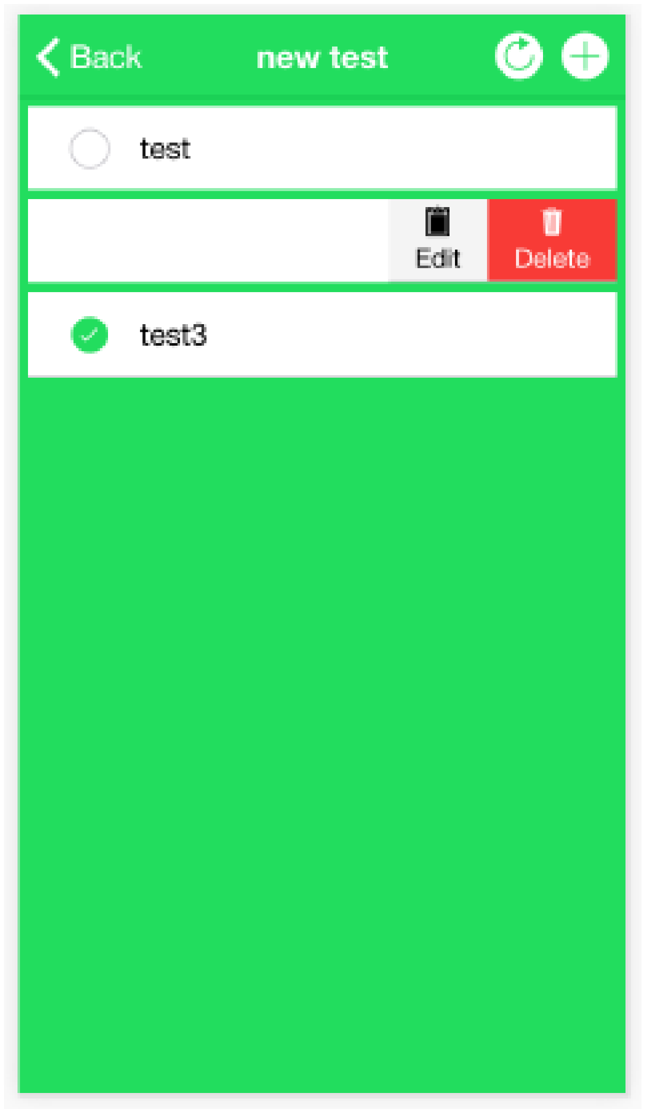
The click handler we attach to the button calls a function `viewChecklist` but it also passes in a parameter of `checklist`. We haven't defined what this is yet, but eventually we will be creating a bunch of these items from an array of data, and we will create a reference to each individual item that we can pass into this function. So eventually, the `checklist` we are passing in here will be a reference to the specific item that was clicked (we will discuss exactly how we do 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 'Edit' and 'Delete' buttons which will also pass in a reference to the checklist it was called on (again, we will have to create this reference later, for now it will just cause issues).

That's all there is to the home page, so let's move on to the checklist page now.

## 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/pages/checklist-page/checklist-page.html` to reflect the following**

```
<ion-header>

    <ion-navbar color="secondary">

        <ion-title>
```

```
            CHECKLIST TITLE
        </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="uncheckItems()"><ion-icon
               name="refresh-circle"></ion-icon></button>
            <button ion-button icon-only (click)="addItem()"><ion-icon
               name="add-circle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>

  <ion-list no-lines>

    <ion-item-sliding>

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

        <ion-item-options>
            <button ion-button icon-only color="light"
               (click)="renameItem(item)"><ion-icon
               name="clipboard"></ion-icon> Edit</button>
            <button ion-button icon-only color="danger"
               (click)="removeItem(item)"><ion-icon name="trash"></ion-icon>
               Delete</button>
        </ion-item-options>
    </ion-item-sliding>

  </ion-list>

</ion-content>
```

You already know how the navbar works, this time we've just got an extra button inside of our `<ion-buttons>` and it is still using the **end** attribute, so both items will be aligned to the right. 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 now, but this time we have an `<ion-checkbox>` as the main item instead of the title of the checklist and when it is clicked it will trigger the `toggleItem()` function which we will need to define later. Notice that we are just using a plain `<ion-item>` now instead of `<button ion-item>`, this is because we're attaching the click handler directly to the checkbox rather than having it on the entire item.

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 modifying 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 it will. 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:

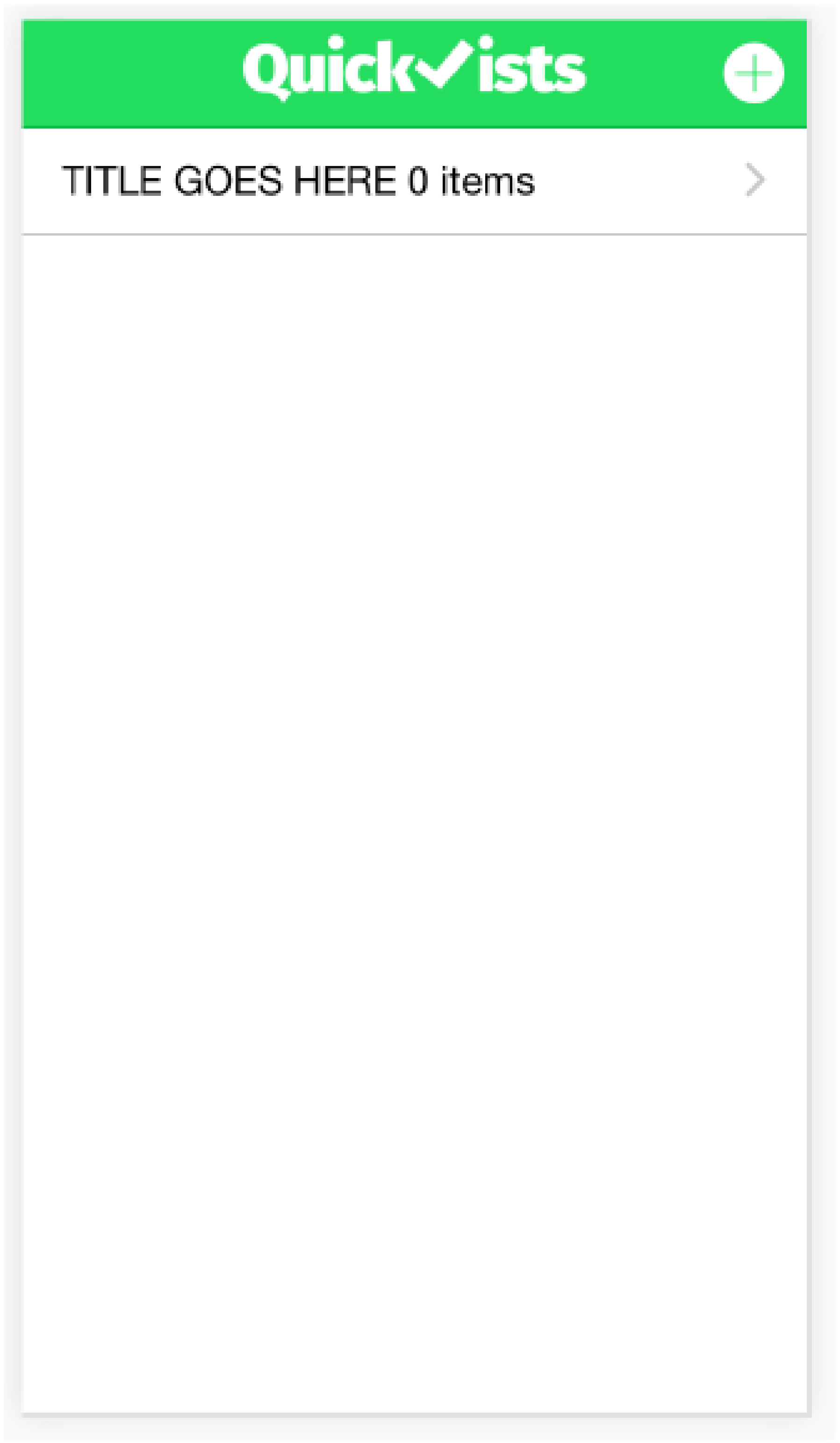
```
<something myName="myName">
```

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 2 looks a little confusing at first, but once you get your head around it it's quite nice to use.

## 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!

# Lesson 5: Creating Checklists and Checklist Items

We've done a lot of setting up and structuring so far, but in this lesson we'll be getting to the bones of what we're building. We'll be adding ways to create new checklists, viewing those checklists and adding items to them (as well as modifying any items or the checklist itself). It's going to be a big one so strap in and get some coffee ready if you are so inclined.

## Checklists

The first thing we are going to do is add everything we need for creating and viewing checklists. This will mean adding to our class definition, as well as modifying the template we created before to actually display the checklist data.

Let's start off by setting up our class definition.

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

```
import { Component } from '@angular/core';
import { NavController, AlertController, Platform } from 'ionic-angular';
import { ChecklistPage } from '../checklist-page/checklist-page';
import { ChecklistModel } from '../../models/checklist-model';
import { Data } from '../../providers/data';

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

  checklists: ChecklistModel[] = [];

  constructor(public nav: NavController, public dataService: Data, public
     alertCtrl: AlertController, public platform: Platform) {

  }

  ionViewDidLoad() {

  }

  addChecklist(): void {

  }

  renameChecklist(checklist): void {

  }

  viewChecklist(checklist): void {

  }

  removeChecklist(checklist): void{

  }

  save(): void{

  }

}
```

We're importing a few things from the Ionic library here. **NavController** you should already be pretty familiar with, but you might not know what **AlertController** is. AlertController allows us to present various alerts to the user, including a basic prompt, prompts with input, confirmation prompts and more. We will be using these as a method to add new checklists.

We're also importing our **ChecklistPage** which we will finish implementing later, but most importantly we are importing the Checklist Model we created in the last lesson.

Finally, we also import the Data provider we generated earlier, but we won't be implementing its functionality until later.

By adding the `public` keyword in the constructor we are simply setting up a reference to the NavController and Data provider that we can use throughout the class later by referencing `this.nav` and `this.dataService`. Basically it's shorthand for this:

```
constructor(nav: NavController, dataService: Data){
  this.nav = nav;
  this.dataService = dataService;
}
```

which you may have seen used before.

We've also declared a `checklists` array at the top of the class which will make it accessible throughout the class by referencing `this.checklists`. The rest of the class is various functions which we will step through creating one by one now.

### addChecklist

This function will be responsible for allowing the user to create a new checklist. It will launch a prompt, and use the data that is entered to create a new checklist (making use of the data model we created).

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

```
  addChecklist(): void {
```

```
  let prompt = this.alertCtrl.create({
    title: 'New Checklist',
    message: 'Enter the name of your new checklist below:',
    inputs: [
      {
        name: 'name'
      }
    ],
    buttons: [
      {
        text: 'Cancel'
      },
      {
        text: 'Save',
        handler: data => {
          let newChecklist = new ChecklistModel(data.name, []);
          this.checklists.push(newChecklist);

          newChecklist.checklist.subscribe(update => {
            this.save();
          });

          this.save();
        }
      }
    ]
  });

  prompt.present();
}
```

We are presenting a prompt to the user that will contain a single `name` input field, and two buttons `Cancel` and `Save`. The cancel button does nothing except dismiss the prompt, but we add a handler to the save button that will pass in the data that was entered into the input field.

Inside of this handler we first generate a new checklist by passing the entered name into a new instance of our checklist model, and then we **push** that object into our `this.checklists` array. Then we subscribe to the observable we added to the data model in the last lesson to listen for whenever the checklist is modified in anyway, and when it is we trigger the save function. Notice that we have two calls to save here, one that is triggered by the observable and one that fires straight away (since we have just added a new checklist).

Finally, we trigger the prompt by calling its `present` method.

If you take a look at your template file again, you'll remember that we've already added a call to this function for when the add button is clicked:

```
<button (click)="addChecklist()"><ion-icon
   name="add-circle"></ion-icon></button>
```

**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): void {

    let prompt = this.alertCtrl.create({
      title: 'Rename Checklist',
      message: 'Enter the new name of this checklist below:',
      inputs: [
        {
          name: 'name'
        }
      ],
      buttons: [
        {
          text: 'Cancel'
        },
        {
          text: 'Save',
          handler: data => {

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

            if(index > -1){
              this.checklists[index].setTitle(data.name);
              this.save();
            }

          }
        }
      ]
    });

    prompt.present();

  }
```

The first thing that you may notice is that it looks very similar to our `addChecklist` function, and that is because it is. We use the same prompt with the same inputs and buttons, we just have a slightly different handler.

Notice that we are passing in a parameter to this function, which will be a reference to the checklist that we want to rename. We will be updating the template shortly to pass in this reference, but for now just pretend that we have it.

We use this reference to the checklist to find it in our `this.checklists` array and then set it to the new title that was entered, then we trigger a save.

Again, if you recall from before, we have already set up a click handler that will call this function in the template:

```
    <button light (click)="renameChecklist(checklist)"><ion-icon
       name="clipboard"></ion-icon></button>
```

### 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 function is quite a lot simpler because it doesn't require any user input, we just need to get rid of the checklist. Just as we did before, we are passing in a reference to the checklist and then finding the checklist in our `this.checklists` array. We then simply remove it from the array using the **splice** method and trigger a save.

Here's the code from the template that triggers this function:

```
  <button danger (click)="removeChecklist(checklist)"><ion-icon
     name="trash"></ion-icon> Delete</button>
```

### viewChecklist

We can create and modify our checklists now, but we also need to be able to see the details of specific checklists, and to add individual items to checklists. To do this, we're going to use our NavController to **push** a new page and pass in a reference to the checklist that was clicked.

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

```
viewChecklist(checklist): void {
  this.nav.push(ChecklistPage, {
    checklist: checklist
  });
}
```

We pass in the **ChecklistPage** we imported before (which we are yet to finish) to the push method, as well as the data we want to send along to the new page, which is a reference to the checklist the user is trying to view. We will be able to use **NavParams** in the class for our checklist page to grab this data later.

### save

This one function is going to be the odd one out for now as we aren't actually going to implement it. There's quite a lot that needs to go into it so we'll be covering saving and loading data in its own lesson later.

To bring everything together, we need to finish off the template for the home page. We can do all of this stuff with our data now, but we can't even see the results.

**> Modify `src/pages/home/home.html` to reflect the following:**

```
<ion-header>
  <ion-navbar color="secondary">
    <ion-title>
      <img src = "assets/images/logo.png" />
    </ion-title>
    <ion-buttons end>
      <button ion-button icon-only (click)="addChecklist()"><ion-icon
         name="add-circle"></ion-icon></button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-list no-lines>

    <ion-item-sliding *ngFor="let checklist of checklists">

      <button ion-item (click)="viewChecklist(checklist)">
          {{checklist.title}}
          <span>{{checklist.items.length}} items</span>
      </button>

      <ion-item-options>
        <button ion-button icon-only color="light"
           (click)="renameChecklist(checklist)"><ion-icon
           name="clipboard"></ion-icon></button>
        <button ion-button icon-only color="danger"
           (click)="removeChecklist(checklist)"><ion-icon
           name="trash"></ion-icon></button>
      </ion-item-options>

    </ion-item-sliding>

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

We're doing a few interesting things here now, most notably we have added an `ngFor` loop:

```
<ion-item-sliding *ngFor="let checklist of checklists">
```

What this will do is loop over every entry we have in our `this.checklists` array and create a sliding item for it in the list. Remember, the `*` syntax used in front of the `ngFor` here is a shortcut for creating embedded templates in Angular 2, so what we are doing essentially is creating a template that is "stamped" out for as many times as we have items in our array. Each time this template is "stamped" it will contain the information for the specific item it was stamped out for, so anywhere inside of the `ngFor` loop we can grab the data of the specific checklist that is being rendered using:

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

Notice that we also have a `let` infront of `checklist` in the `ngFor` loop. In Angular 2 using `let` like this allows us to create a local variable, and this is what then allows us to pass a reference to the specific checklist into all of the functions we just created. To make the concept more clear, if we were to use the following code instead:

```
<ion-item-sliding *ngFor="let check of checklists">
```

Then we would render the data like this:

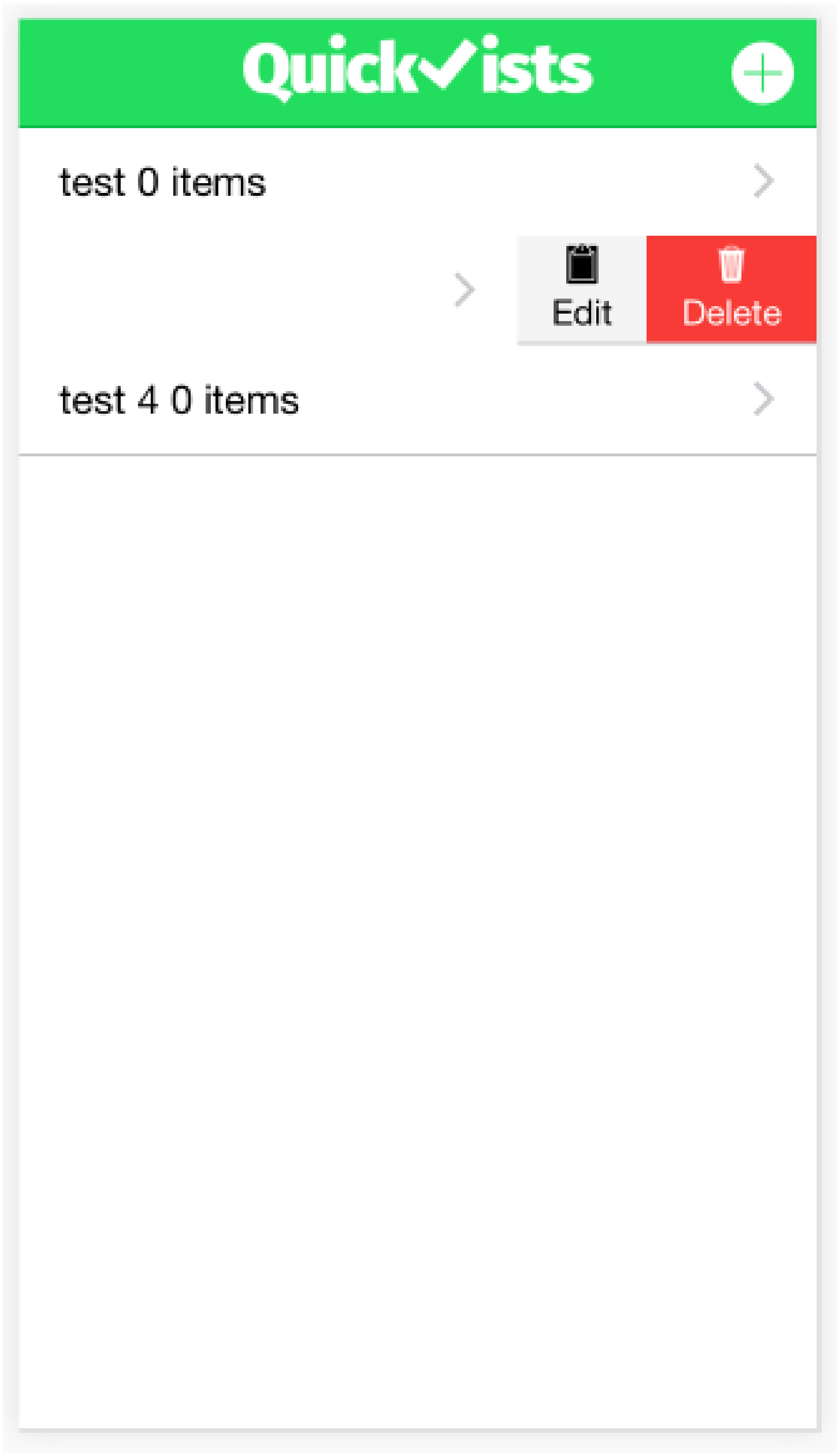
```
{{check.title}}
```

and pass the reference into our functions like this:

```
removeChecklist(check)
```

That just about finishes up our home page, and you should now be able to add, edit and delete checklists, as well as launch the details page for a specific checklist (which won't contain anything just yet, and won't really work).

If you fire up `ionic serve` now you should see something like this:



test 0 items

test 4 0 items

It is important to note that **you must remove this line from checklist.html**:

```
    <ion-checkbox [checked]="item.checked"
```

```
        (click)="toggleItem(item)"></ion-checkbox>
```

**if you want to create an item and then go to its detail page**. In a later lesson we will only include this if there is item data available, but right now we aren't doing that so an error will be caused since it is trying to access data that does not exist.

We will get to all of this later anyway, so you only need to do these extra steps if you want to have a play around with the app right now.

## Checklist Items

Now that we can trigger the checklist detail page, we'd best get some content in there and add a way for people to create and modify individual checklist items. In this section we will be implementing our Checklist Page, which is launched by the home page and is supplied some data regarding which checklist is being viewed.

Let's start by setting up the class definition again.

**> Modify `src/pages/checklist-page/checklist-page.ts` to reflect the following:**

```
import { Component } from '@angular/core';
import { NavController, NavParams, AlertController } from 'ionic-angular';

@Component({
  selector: 'checklist-page',
  templateUrl: 'checklist-page.html'
})
export class ChecklistPage {

  checklist: any;

  constructor(public nav: NavController, public navParams: NavParams, public
      alertCtrl: AlertController){
    this.checklist = this.navParams.get('checklist');
  }

  addItem(): void {

  }

  toggleItem(item): void {

  }

  removeItem(item): void {

  }

  renameItem(item): void {

  }

  uncheckItems(): void {

  }
}
```

There’s nothing much going on here that you are not already familiar with, the only thing out of the ordinary is the use of **NavParams**. When we pass in data to another page, we can grab it by injecting NavParams and using the `get` method. In this instance we are just passing in the checklist data that we want to view, but you can also pass in multiple values if you like.

Just as we did before, we are going to go through implementing these functions one by one now. A lot of these will be quite similar to what we just did for the home page.

**addItem**

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

```
addItem(): void {

    let prompt = this.alertCtrl.create({
      title: 'Add Item',
      message: 'Enter the name of the task for this checklist below:',
      inputs: [
        {
          name: 'name'
        }
      ],
      buttons: [
        {
          text: 'Cancel'
        },
        {
          text: 'Save',
          handler: data => {
                   this.checklist.addItem(data.name);
          }
        }
      ]
    });

    prompt.present();
```

```
  }
```

This should all look very familiar, but notice the difference in the handler. Since we created an **addItem** help function on our data model, all we have to do is call that and pass in the name of the item we want to create (I told you the data model would come in handy!).

**renameItem**

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

```
  renameItem(item): void {

      let prompt = this.alertCtrl.create({
        title: 'Rename Item',
        message: 'Enter the new name of the task for this checklist below:',
        inputs: [
          {
            name: 'name'
          }
        ],
        buttons: [
          {
            text: 'Cancel'
          },
          {
            text: 'Save',
            handler: data => {
              this.checklist.renameItem(item, data.name);
            }
          }
        ]
      });

      prompt.present();

  }
```

Once again, almost the exact same idea except we are calling the **renameItem** helper function on our data model in the handler, and we are also passing through a reference to the specific item that we are renaming.

### removeItem

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

```
    removeItem(item): void {
        this.checklist.removeItem(item);
    }
```

This one is even simpler, we simply call the `removeItem` helper function on the data model and pass it a reference to the item we want to delete.

### toggleItem

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

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

This function is used to toggle the checkmarks on an individual item on and off, and once more we simply pass through a reference to the item we want to toggle to the data model.

### uncheckItems

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

```
    uncheckItems(): void {
        this.checklist.items.forEach((item) => {
            if(item.checked){
                this.checklist.toggleItem(item);
            }
        });
    }
```

This function is tied to the reset button we added to our template, and will loop through every item we have in the checklist and call the `toggleItem` function in the data model if the current item is checked. This allows the user to uncheck all items at once.

Now all we have left to do is update the template for our checklist page. We've already set up most of the structure for this template, but just like with the home page we will need to add a little bit more to handle displaying data.

**> Modify `src/pages/checklist-page/checklist-page.html` to reflect the following**

```
<ion-header>
  <ion-navbar color="secondary">
    <ion-title>
        {{checklist.title}}
    </ion-title>
    <ion-buttons end>
        <button ion-button icon-only (click)="uncheckItems()"><ion-icon
           name="refresh-circle"></ion-icon></button>
        <button ion-button icon-only (click)="addItem()"><ion-icon
           name="add-circle"></ion-icon></button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content>

  <ion-list no-lines>

    <ion-item-sliding *ngFor="let item of checklist.items">

        <ion-item>
        <ion-label>{{item.title}}</ion-label>
            <ion-checkbox [checked]="item.checked" (click)="toggleItem(item)">
            </ion-checkbox>
        </ion-item>

        <ion-item-options>
            <button ion-button icon-only color="light"
               (click)="renameItem(item)"><ion-icon
               name="clipboard"></ion-icon></button>
            <button ion-button icon-only color="danger"
               (click)="removeItem(item)"><ion-icon
               name="trash"></ion-icon></button>
        </ion-item-options>
    </ion-item-sliding>

  </ion-list>

</ion-content>
```

This should all look pretty similar to the home page template, but there are a few differences. We are using the checklist data from our class to display the title of the current checklist in the navbar. We are again looping through data using `ngFor` but this time we are only looping through the items, which are a child of the checklist. Also notice that as well as rendering data using double braces like this:

```
{{item.title}}
```

we can also set properties on elements using the square brackets like this:

```
[checked]="item.checked"
```

this will set **checked** to be the value of whatever `item.checked` evaluates to.

## Summary

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 and adding your own checklists and checklist items.

In the next lesson we'll work on saving data and then we'll look into prettying up the application (function before form right?).

# 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're subscribing to our Observables and calling the `save` function every time some data changes, we just need to implement that function now.

**> Modify the save function in `src/pages/home/home.ts` to reflect the following:**

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

If you recall, earlier we already generated and imported a data service, so all we need to change here is to add a call to it and pass in the current checklists data. Of course, we haven't actually implemented that data service yet so it won't do anything with the data, so let's fix that now.

## Saving Data

We're going to add the code for **data.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/providers/data.ts` to reflect the following:**

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

@Injectable()
export class Data {
  constructor(public storage: Storage){

  }

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

  save(data): void {

    let saveData = [];

    //Remove observables
    data.forEach((checklist) => {
      saveData.push({
        title: checklist.title,
        items: checklist.items
      });
    });

    let newData = JSON.stringify(saveData);
    this.storage.set('checklists', newData);
  }
}
```

There's a new import here that we haven't seen before:

```
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('checklists');
}
```

This function will allow us to retrieve the latest data that has been stored, and it will return it in the form of a **Promise**. We are setting the return type for this function as a Promise that returns `<any>` type, this is one of the more complicated types. Remember, adding types like this is not required so if you are confused by this and would prefer to leave it out you could just do this instead:

```
getData(){
    return this.storage.get('checklists');
  }
```

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 (which will be a promise which resolves with the data that is currently in storage). Remember that this operation is not instant, and 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 `saveData` function, which handles actually saving the data into storage:

```
save(data): void {
    let saveData = [];

    //Remove observables
    data.forEach((checklist) => {
      saveData.push({
        title: checklist.title,
        items: checklist.items
      });
    });

    let newData = JSON.stringify(saveData);
    this.storage.set('checklists', newData);
  }
```

As I mentioned, we are storing the data as a single JSON encoded string, so we call the `JSON.stringify` function and then store the data using the `set` method on our storage object. Before we do that though, we remove the observable stuff from the data by only pushing the `title` and the `items` since it doesn't play nice with JSON (it causes a circular object error), and we'll just be recreating them later anyway.

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.

## Loading Data

We are going to want to load the checklists data from storage whenever the user opens the application, so a good place to do this is the `ionViewDidLoad` hook on our home page, which triggers as soon as the page is loaded. We are also going to make a slight modification to the constructor.

**> Modify the `constructor` and `ionViewDidLoad` in `src/pages/home/home.ts` to reflect the following:**

```
constructor(public nav: NavController, public dataService: Data, public
    alertCtrl: AlertController, public storage: Storage, public platform:
    Platform) {

}

ionViewDidLoad(){

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

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

      let savedChecklists: any = false;

      if(typeof(checklists) != "undefined"){
        savedChecklists = JSON.parse(checklists);
      }

      if(savedChecklists){

        savedChecklists.forEach((savedChecklist) => {

          let loadChecklist = new ChecklistModel(savedChecklist.title,
              savedChecklist.items);
          this.checklists.push(loadChecklist);

          loadChecklist.checklist.subscribe(update => {
            this.save();
          });
          });

        }

      });

    });

  }
```

We're making a call to the `getData` function that we just defined in our data service. As I mentioned, the getData function returns a promise rather than the data directly, which allows us to handle the response here once it has finished loading. If the getData function just returned the data directly, rather than a promise, then the data would likely not have even been returned yet when we try to access it.

So we wait for the data to be retrieved, and then pass the checklists data into our handler. First we decode the JSON string into an array that we can work with, and then we loop through every item in the array and create a new Checklist Model based on it's data. The reason we loop through the data and create new models rather than just settings `this.checklists` to be `savedChecklists` directly is because by converting the checklists into a JSON string when we store it we lose the ability to use the helper functions we defined on the model. So we just use the title and items data to recreate new objects for all the checklists.

Finally, we set up the listener for the Observable again so that the save function will be triggered whenever the data changes. It's important that we do all of this within the `platform.ready()` call, which will only execute after the device is ready. Since we are interacting with the devices storage, if we try to do that before the device is ready it will cause some issues.

## Summary

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 & 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.

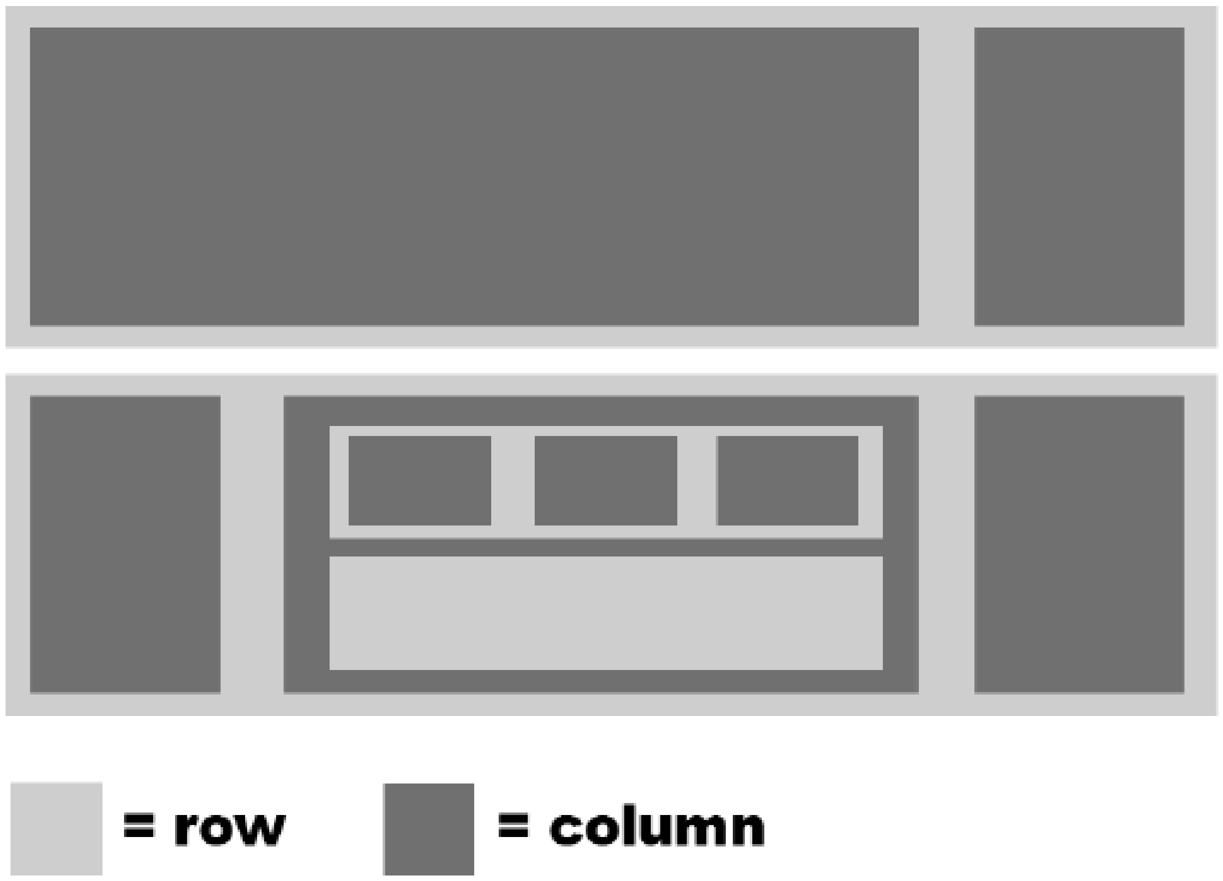
**> Modify `src/pages/intro-page/intro-page.html` to reflect the following**

```
<ion-content>

  <ion-slides [options]="slideOptions">

    <ion-slide>
      <img src="assets/images/slide1.png" />
    </ion-slide>

    <ion-slide>
      <img src="assets/images/slide2.png" />
    </ion-slide>

    <ion-slide>
      <img src="assets/images/slide3.png" />
    </ion-slide>

    <ion-slide>
      <ion-row>
        <ion-col>
          <button ion-button color="light" (click)="goToHome()"
              style="margin-top:20px;">Start Using Quicklists</button>
        </ion-col>
      </ion-row>
      <ion-row>
        <ion-col>
          <img src="assets/images/slide4.png" />
        </ion-col>
      </ion-row>
    </ion-slide>

  </ion-slides>

</ion-content>
```

The first thing you might notice about this is that it doesn't contain a navbar, only the content area. It's not necessary to include the navigation bar 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 **options** property so that we can configure it with some options in our class definition in a moment, 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 2'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:

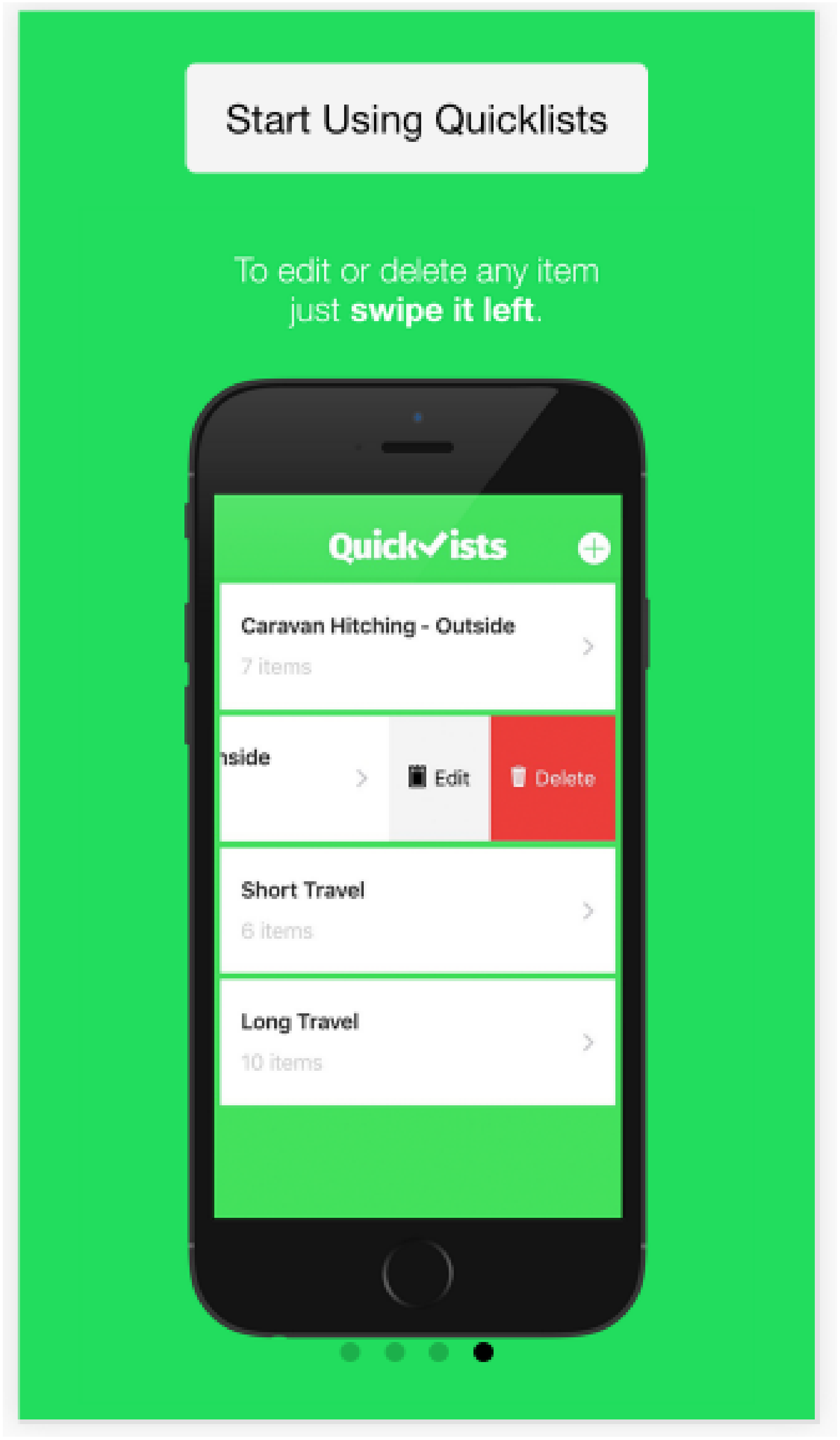


This is a very simple example because we just want the button to appear above the image, but you can create quite complex layouts by supplying a width to the cols like this:

```
<ion-row>
    <ion-col width-10></ion-col>
    <ion-col width-50></ion-col>
```

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

and you can include as much nesting as you like. The result of our grid layout will look like this:



Now we need to define the class for our intro component.

**> Modify `src/pages/intro-page/intro-page.ts` to reflect the following**

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

@Component({
  selector: 'intro-page',
  templateUrl: 'intro-page.html'
})
export class IntroPage {

  slideOptions: any;

  constructor(public nav: NavController){
    this.slideOptions = {
      pager: true
    };
  }

  goToHome(): void {
    this.nav.setRoot(HomePage);
  }
}
```

Isn't this just the simplest class you've seen so far? All it does is import the home page, set the pager option for our slider, and change the root page to it using the NavController when the `goToHome` function is called. This allows us to tap the button on the last slide to go to our main home page view, but we will have a bit of a problem. Every time the user opens the application they are going to have to go through this tutorial to get to the main app. To solve this, we are going to make one more change to our **home.ts** file.

**> Modify `src/pages/home/home.ts` to reflect the following imports:**

```
import { Component } from '@angular/core';
import { NavController, AlertController, Platform } from 'ionic-angular';
import { ChecklistPage } from '../checklist-page/checklist-page';
import { ChecklistModel } from '../../models/checklist-model';
import { Data } from '../../providers/data';
import { IntroPage } from '../intro-page/intro-page';
import { Storage } from '@ionic/storage';
```

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

```
  checklists: ChecklistModel[] = [];

  constructor(public nav: NavController, public dataService: Data, public
     alertCtrl: AlertController, public storage: Storage, public platform:
     Platform) {

  }

  ionViewDidLoad(){

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

      this.storage.get('introShown').then((result) => {
        if(!result){
          this.storage.set('introShown', true);
          this.nav.setRoot(IntroPage);
        }
```

```
   });

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

     let savedChecklists: any = false;

     if(typeof(checklists) != "undefined"){
       savedChecklists = JSON.parse(checklists);
     }

     if(savedChecklists){

       savedChecklists.forEach((savedChecklist) => {

         let loadChecklist = new ChecklistModel(savedChecklist.title,
             savedChecklist.items);
         this.checklists.push(loadChecklist);

         loadChecklist.checklist.subscribe(update => {
           this.save();
         });

       });

     }

   });

 });
```

```
  }
```

We’re importing and making use of the Storage service again now. We’re going to use this to store a flag that will tell us whether or not the tutorial has already been viewed.

So we set up our new storage and we check for the existence of an `introShown` flag. If it does not exist then we switch to our intro tutorial page and then set that flag to be true so it doesn’t show next time.

**NOTE:** For testing purposes, if you want to clear this flag you will need to delete the WebSQL database that is created in your browser. On Chrome, you can do that by going to `chrome://settings/cookies`, searching for `localhost`, and then deleting `_ionicstorage`. This will of course delete all data that is stored in the WebSQL database, not just the `introShown` flag.

## 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’s quite a few different ways we can add styles to the application. We will be basically using all of these methods. We will be adding specific styles to each of our components, we will be adding some generic styles in our core file, and we will be overriding some SASS variables in the variables file. 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.

Since we use SASS for CSS in Ionic 2, we are able to nest CSS styles. Since each page is its own component, we can easily make sure we only target elements that are within that component. Even though we have individual `.scss` files for each component, the CSS rules still apply globally. So if you were to add the following style to one of your components:

```
p {
  font-size: 1.2em !important;
}
```

It would apply to every component in the application.

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

```
intro-page {


  ion-slide {
    background-color: #32db64;
  }


  ion-slide img {
    height: 85vh !important;
    width: auto !important;
  }


}
```

As you can see, we have defined the styles for the page inside of `intro-page`, so the styles will only apply to elements inside of the `<intro-page>` element which will be added to the DOM (because this component has a `selector` of `intro-page`), not the entire application. 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.

The rest of our components are going to require us adding some classes into our template that we can hook into, so from now on I'll post both the finalised template code, as well as the styles to go along with it.

**> Modify `src/pages/home/home.html` to reflect the following:**

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

```
    <ion-title>
      <img src = "assets/images/logo.png" />
    </ion-title>
    <ion-buttons end>
      <button ion-button icon-only (click)="addChecklist()"><ion-icon
          name="add-circle"></ion-icon></button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-list no-lines>

    <ion-item-sliding *ngFor="let checklist of checklists">

      <button ion-item (click)="viewChecklist(checklist)" class="home-item">
          {{checklist.title}}
          <span class="secondary-detail">{{checklist.items.length}}
             items</span>
      </button>

      <ion-item-options>
        <button ion-button icon-only color="light"
           (click)="renameChecklist(checklist)"><ion-icon
          name="clipboard"></ion-icon></button>
        <button ion-button icon-only color="danger"
           (click)="removeChecklist(checklist)"><ion-icon
          name="trash"></ion-icon></button>
      </ion-item-options>
    </ion-item-sliding>


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

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

```
home-page {


  ion-item-sliding {
    margin: 5px;
  }


  .home-item {
      font-size: 1.2em;
      font-weight: bold;
      color: #282828;
      padding-top: 10px;
      padding-bottom: 10px;
  }


  .secondary-detail {
    display: block;
    color: #cecece;
    font-weight: 400;
    margin-top: 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/pages/checklist-page/checklist-page.scss` to reflect the following:**

```
checklist-page {


  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;
  }


  ion-checkbox .item-inner {
    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 styles to `src/app/app.scss`:**

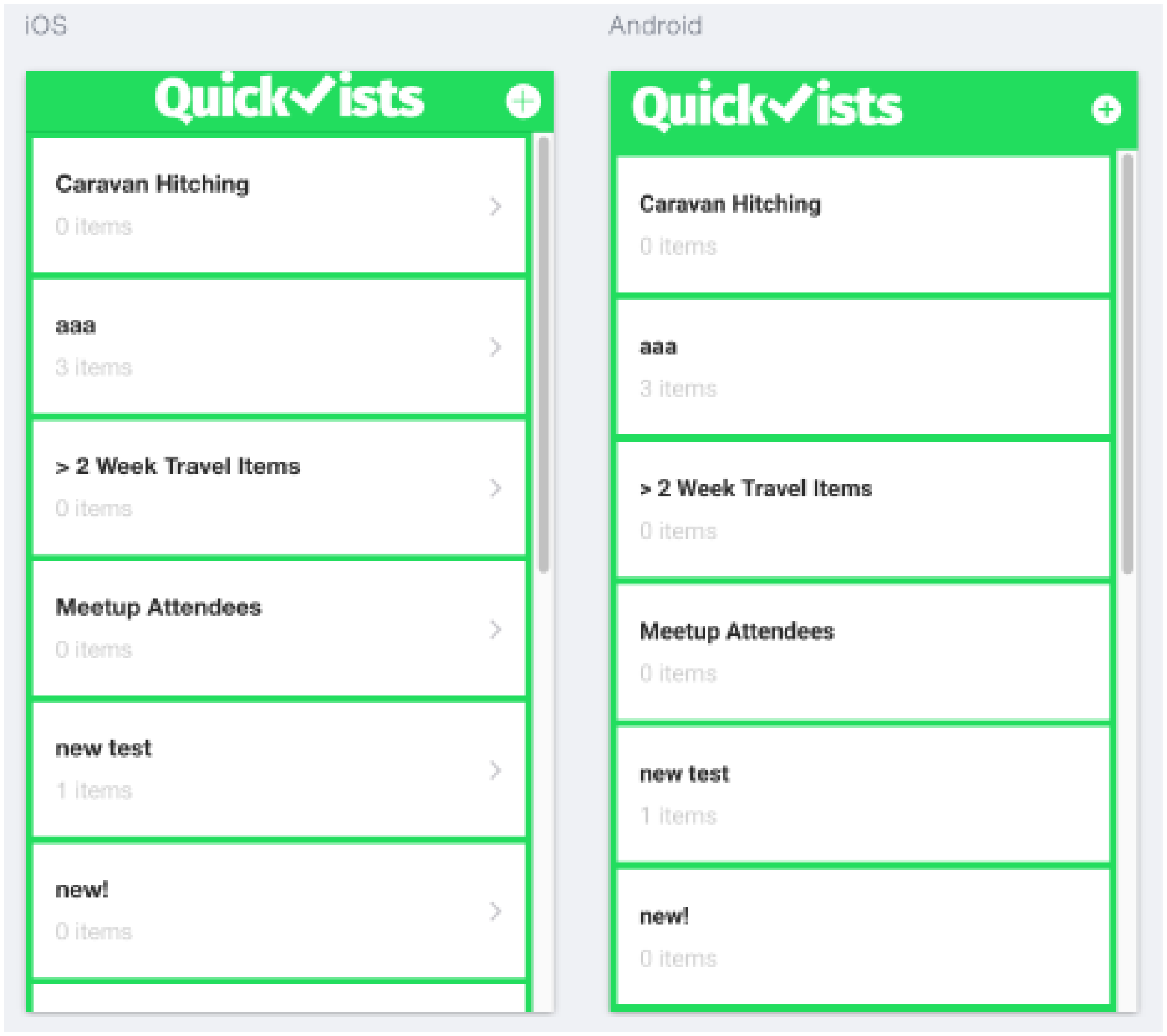
```
ion-content {
    background-color: #32db64 !important;
}

.logo {
    max-height: 39px;
}

button {
    border: none !important;
}
```

Here we're making the entire application have a background colour of green, we set a maximum height for the logo and remove borders from our buttons. Finally, we are going to override some of the SASS variables.

**> Modify the `Named Color Variables` section in `src/theme/variables.scss` to reflect the following:**

```
$colors: (
  primary:    #387ef5,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222,
  favorite:   #69BB7B
);

$list-background-color: #fff;
$list-ios-activated-background-color: #3aff74;
$list-md-activated-background-color: #3aff74;

$checkbox-ios-background-color-on: #32db64;
$checkbox-ios-icon-border-color-on: #fff;

$checkbox-md-icon-background-color-on: #32db64;
$checkbox-md-icon-background-color-off: #fff;
$checkbox-md-icon-border-color-off: #cecece;
$checkbox-md-icon-border-color-on: #32db64;
```

We've modified the colours for the application and set a few iOS and Android specific styles (the names should make it pretty clear what is being changed). One of the coolest things about Ionic 2 is how well it handles UI differences between iOS and Android, for the most part it just works flawlessly out of the box. If you take a look at the application on iOS and on Android, you will see the differences:



**NOTE:** A handy way to see iOS and Android side by side is to use Ionic Lab, which can be activated by using the `ionic serve -l` command. You may notice that the applications have a scrollbar on the edge when viewing through Ionic Lab, I believe this is a bug and it is not present when running on an actual device (or on the Chrome Dev Tools emulator).

As you can see, Ionic 2 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 tutorial. This application is a great example for beginners to start getting their feet wet, and the main take aways 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 observables
- How to navigate and pass data between pages
- How to create a data model

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:

- Retheme the application with your own styling, try different colours, padding, margins and so on **[EASY]**
- Add a 'Date Created' field to the data model that records when a checklist was created, and display it in the template (don't forget to make sure it gets loaded from memory too!) **[MEDIUM]**
- Figure out how many items have been marked as completed in a single checklist, and display a progress indicator (i.e 5/7 completed) **[MEDIUM]**
- Add the ability to attach notes to any specific checklist item **[HARD]**

Remember, the Ionic 2 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.

# Chapter 4

# Testing & Debugging

# 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 2 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.

## 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 on an actual iOS device you will first need to run the following command:

```
npm -g install ios-sim ios-deploy
```

This will install both the iOS Simulator (which will allow you to emulate an iOS device on your computer) and `ios-deploy` will allow you to deploy the application directly to your device when it is attached via USB.

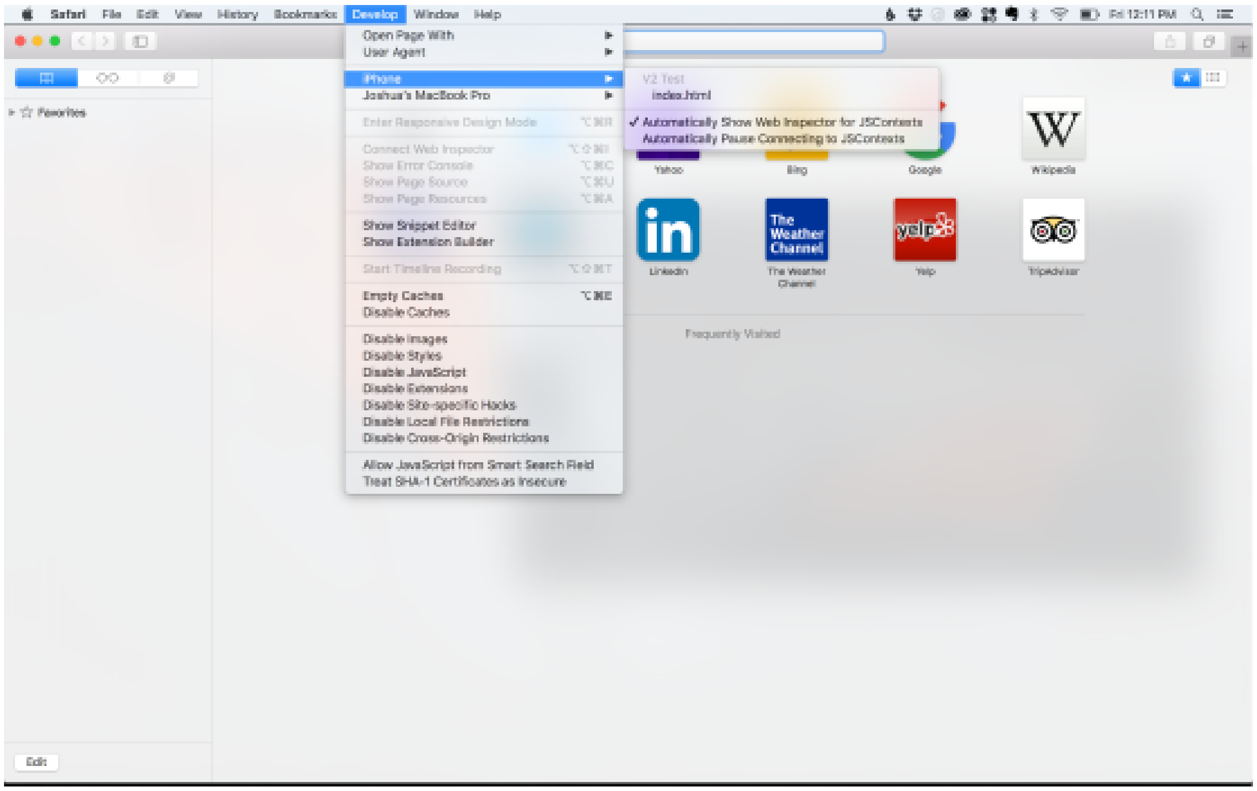
Once that is set up, all you have to do is run:

```
ionic run ios
```

from within your project directory and the application will be deployed to your device (or to the emulator if a device is not available). If it is the first time you have run this command, you may need to run it twice before it will work.

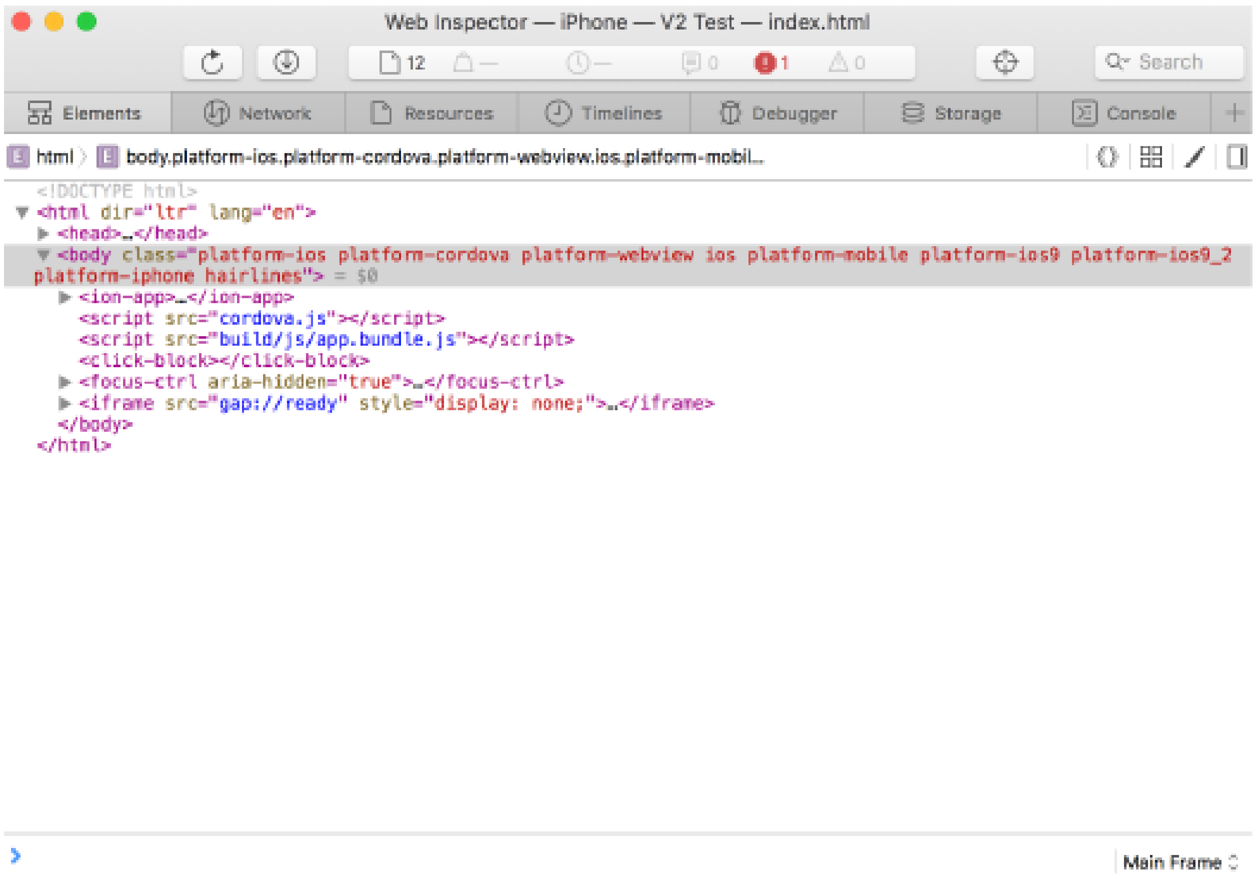
Once the application is running on your device, you can debug it using the Safari Dev Tools on your computer. Simply open up **Safari** and then go to:

```
Develop > iPhone > index.html
```



If the Develop menu does not appear for you, you may first have to enable it by going to `Safari > Preferences > Advanced > Show Develop menu in menu bar.`

Once you open up **index.html** in Safari you should see a debugging screen like this:


Web Inspector — iPhone — V2 Test — index.html
12
1
0
0
Search
Elements
Network
Resources
Timelines
Debugger
Storage
Console
html
body.platform-ios.platform-cordova.platform-webview.ios.platform-mobil...
Main Frame

```
<!DOCTYPE html>
▼ <html dir="ltr" lang="en">
  ▶ <head>…</head>
  ▼ <body class="platform-ios platform-cordova platform-webview ios platform-mobile platform-ios9 platform-ios9_2
  platform-iphone hairlines"> = $0
    ▶ <ion-app>…</ion-app>
      <script src="cordova.js"></script>
      <script src="build/js/app.bundle.js"></script>
      <click-block></click-block>
    ▶ <focus-ctrl aria-hidden="true">…</focus-ctrl>
    ▶ <iframe src="gap://ready" style="display: none;">…</iframe>
    </body>
  </html>
```

**IMPORTANT:** This method will only work if you have a Mac. If you do not have a Mac then you can use GapDebug to install and debug iOS applications. In order to do this you will need to generate an `.ipa` of your application first (we will discuss how to do this in the **Building for iOS** lesson) and install that directly.

It's also a good idea to test the final build of your application using **Test Flight**.

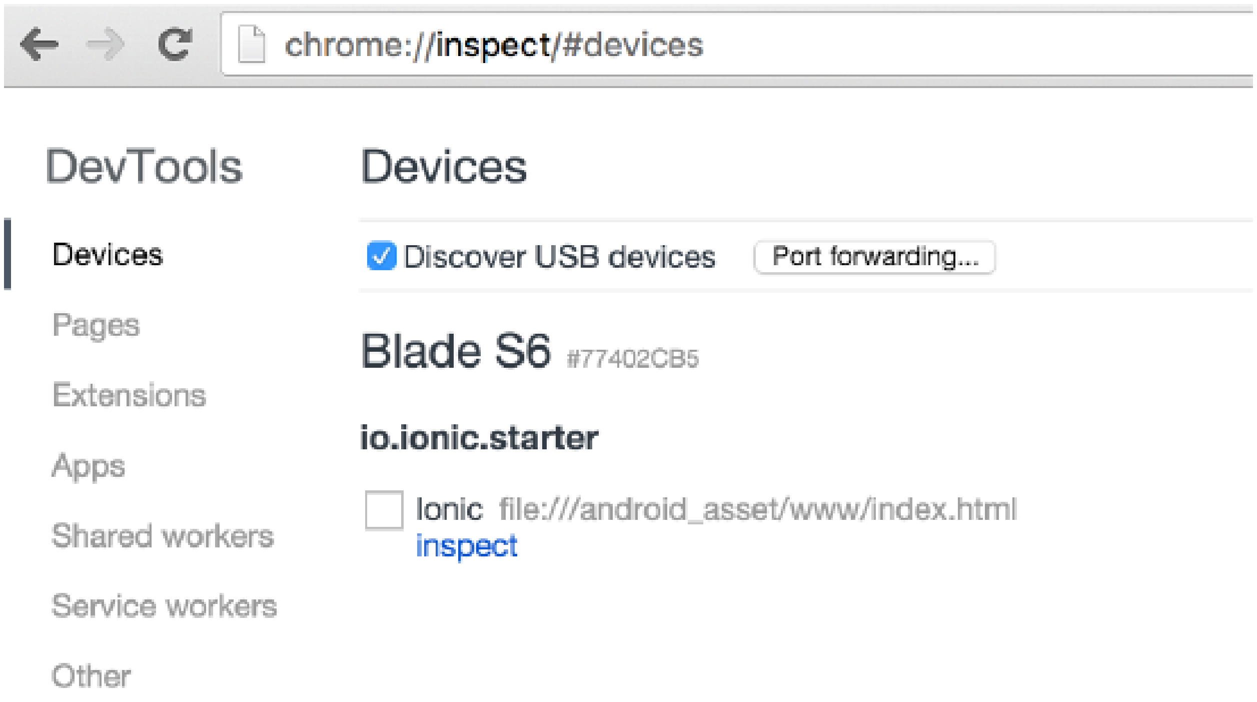
## Android Debugging

Debugging on Android is as simple as attaching the device you want to debug on and then running:

```
ionic run android
```

on your desktop browser (in Chrome) you will now be able to go to the following address:

```
chrome://inspect/#devices
```


chrome://inspect/#devices

DevTools

Devices

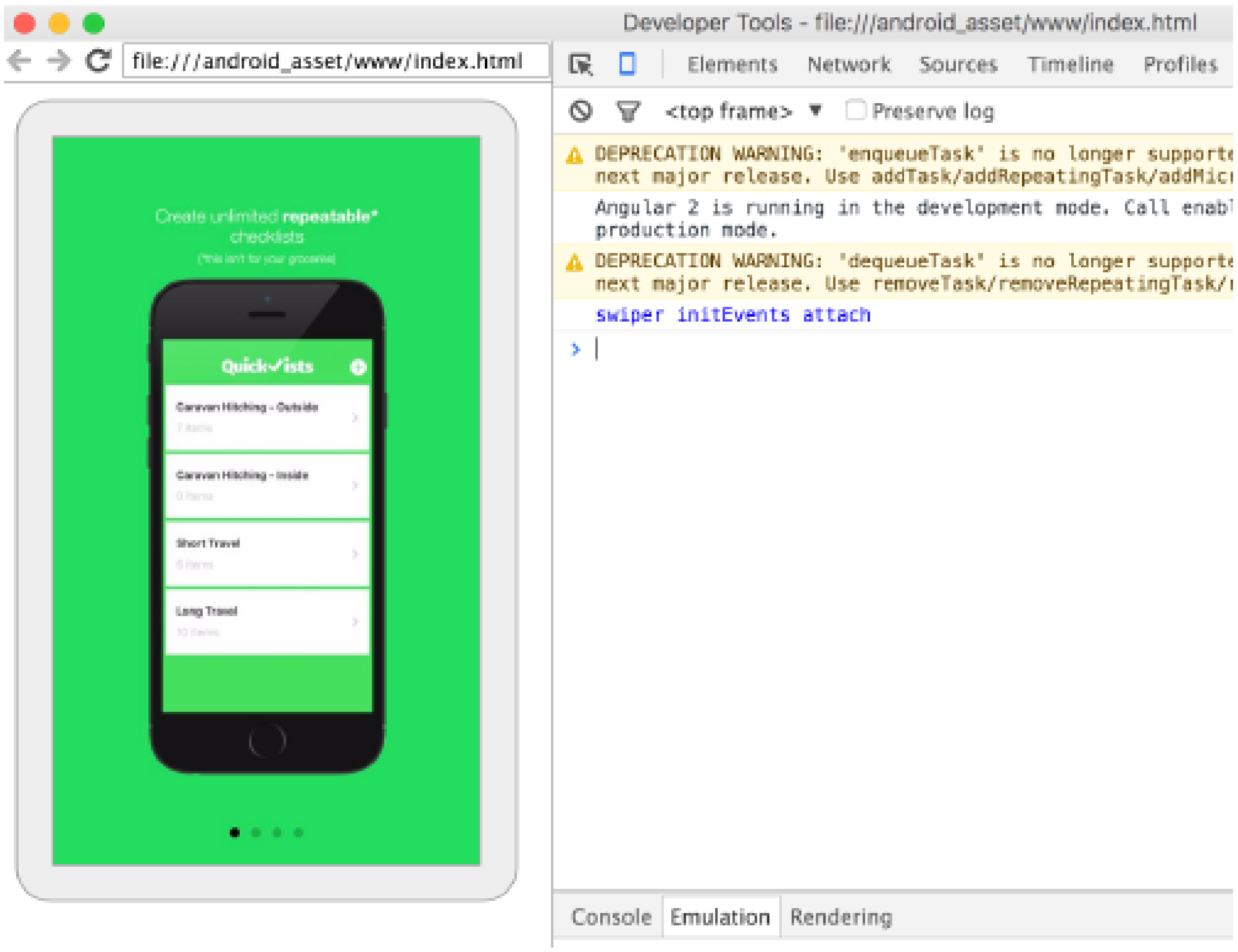
☑ Discover USB devices

Blade S6 #77402CB5

io.ionic.starter

Ionic file:///android_asset/www/index.html

and then click ‘Inspect’ on the device you want to debug on, and you will see the debugging tools:


Developer Tools - file:///android_asset/www/index.html
file:///android_asset/www/index.html
Elements
Network
Sources
Timeline
Profiles
<top frame>
Preserve log
Create unlimited repeatable* checklists
(this isn't for your groceries)
Quicklists
Caravan Hitching - Outside
Caravan Hitching - Inside
Short Travel
Long Travel
Console
Emulation
Rendering

```
DEPRECATION WARNING: 'enqueueTask' is no longer supporte
next major release. Use addTask/addRepeatingTask/addMic
Angular 2 is running in the development mode. Call enab
production mode.
DEPRECATION WARNING: 'dequeueTask' is no longer supporte
next major release. Use removeTask/removeRepeatingTask/
swiper initEvents attach
>
```

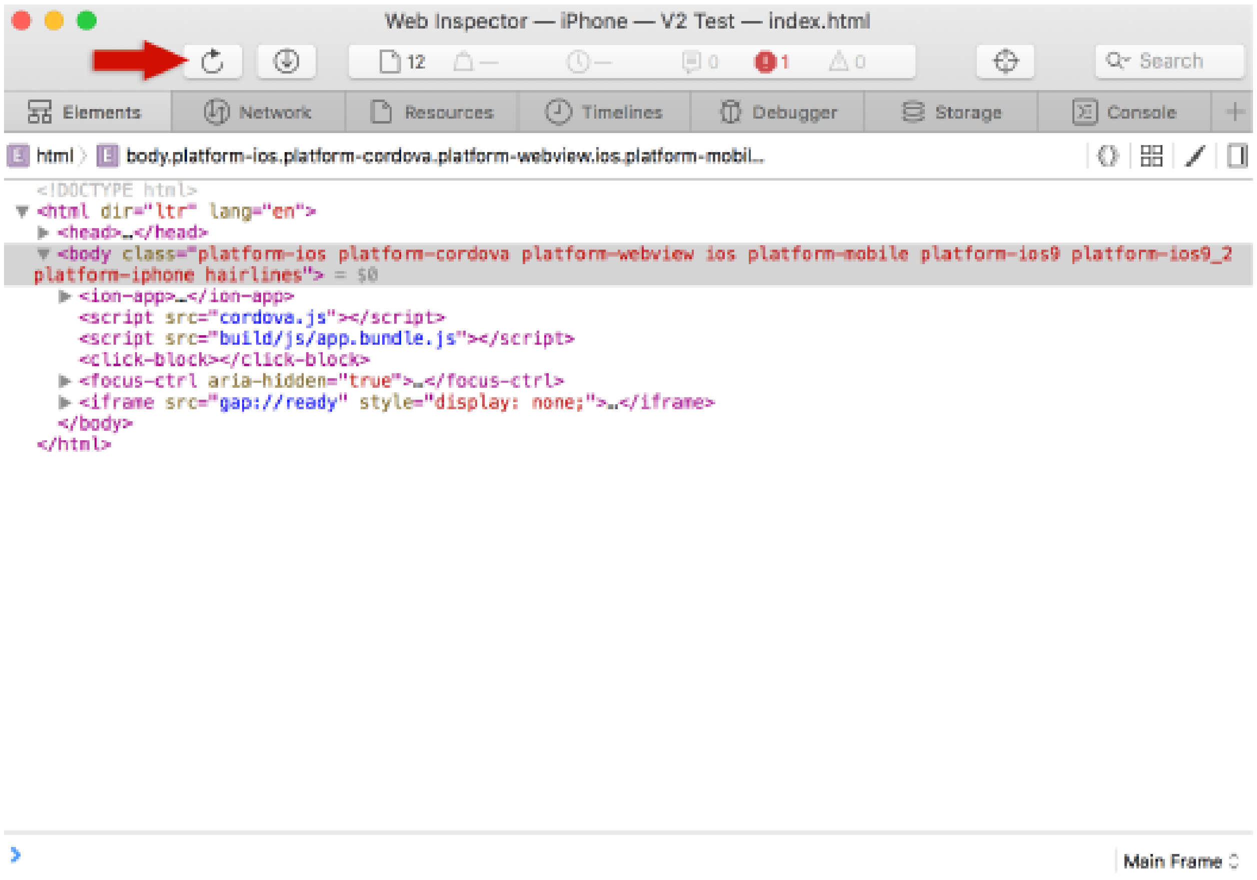
**IMPORTANT:** You will need to have USB debugging enabled on your device for this to work. Enabling USB debugging is different for different devices, so just Google "[YOUR DEVICE] enable usb debugging".

## Tips & Common Errors

The ease of which you can debug applications usually comes from the debugging prowess you develop over years of facing frustrating errors, but there are a few tips and gotchas I can offer that may help you out when debugging Ionic 2 application.

**Facing a white or blank screen at launch?**

This one happens a lot and the reason is usually a startup error when running on a device. Sometimes you will launch your application and just see a blank screen, and even if you open your debugging tools you might not see any errors. This can be because an error occurs before the debugger has launched, which is breaking the application. So if you're running into a problem like this, make sure to get the debugger up and then hit the refresh button from the debugger, which in Safari looks like this:


Web Inspector — iPhone — V2 Test — index.html
12
0
1
0
Search
Elements
Network
Resources
Timelines
Debugger
Storage
Console
html
body.platform-ios.platform-cordova.platform-webview.ios.platform-mobil...
Main Frame

```
<!DOCTYPE html>
▼ <html dir="ltr" lang="en">
  ▶ <head>…</head>
  ▼ <body class="platform-ios platform-cordova platform-webview ios platform-mobile platform-ios9 platform-ios9_2
platform-iphone hairlines"> = $0
    ▶ <ion-app>…</ion-app>
      <script src="cordova.js"></script>
      <script src="build/js/app.bundle.js"></script>
      <click-block></click-block>
    ▶ <focus-ctrl aria-hidden="true">…</focus-ctrl>
    ▶ <iframe src="gap://ready" style="display: none;">…</iframe>
    </body>
  </html>
```

This will reload the application with the debugger already active, which means you're not going to miss any errors that occur right at the start. Nine times out of ten you will discover there is some Javascript error and after fixing that your application should work.

### Debugger

Hopefully you know about how useful breakpoints are for debugging your application, but if not I'll reiterate that you should read this.

You can just open up the 'Sources' tab and put some breakpoints in manually, but you can also simply add this little keyword:

```
debugger;
```

anywhere in your code, and it will cause your application to pause at that point (as long as you have the Dev Tools open).

### 404 Errors

Is your app working fine through the browser but you're getting 404 errors when running on a device? Make sure you have included the Whitelist plugin:

```
ionic plugin add cordova-plugin-whitelist
```

and that you have an appropriate Content Security Policy (CSP) meta tag defined in your **index.html** file:

```
<meta http-equiv="Content-Security-Policy" content="font-src * data:; img-src
   * data:; default-src * 'unsafe-eval' 'unsafe-inline'">
```

### Plugins not working on a device

Of course the first thing to make sure you've done is to run the install command for the plugin:

```
ionic plugin add [plugin name]
```

It might sound silly but make sure to double check with the following command:

```
ionic plugin ls
```

this will list all plugins that are currently installed, and I still even forget to install plugins sometimes. If the plugin is installed then you should also make sure that you are not trying to access the plugin too early. If you try to access plugin functionality before the device is ready, i.e. before this fires:

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

});
```

then it will not work. Finally, if you've exhausted all other options and it is still not working then you should try removing the plugin:

```
ionic plugin rm [plugin name]
```

and then readding the plugin:

```
ionic plugin add [plugin name]
```

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.

## Installing your Application with GapDebug

If you are using PhoneGap Build and have a **.ipa** or **.apk** file you want to test then you will want to use GapDebug. There are other ways to install these files - you could use iTunes to install the .ipa file or you could use **adb** to install the .apk file - but it is so much easier with GapDebug and GapDebug also provides you with some awesome debugging tools (although, if you're looking for some more advanced debugging I would recommend using **adb** as well). To get GapDebug just head over to the website and download it. Once you have it installed, it should be available in your system tray.

After you have installed GapDebug, make sure you read through the **GapDebug First-Time Configuration and Setup** - there is a few things you'll need to do before GapDebug will work properly with your iOS or

Android device.

Once you have everything set up you will be able to connect your device to your computer, open up GapDebug, then just drag and drop the application file (.ipa or .apk) into the device through GapDebug. Not only is this a super quick way to get the application on your device but it also provides you with some extremely useful debugging tools (which are essentially the same as the browser debugging tools you may already be familiar with). Once you have installed your application with GapDebug, just click the application in GapDebug to reveal the debugging tools.

# Chapter 5

# Building & Submitting

# Preparing Assets

At this point you should have your application built, working, and ready for submission to the app store. There's just a couple things we are going to do beforehand though to make sure our application is in "ship shape" before shipping it off to the app stores.

You may be following along and building your own application whilst reading this book, but chances are you've been building one of the example applications. In that case, I'd still recommend you go through these steps, or at least read them, so that you learn how the submission process works. I do however ask that you stop short of *actually* submitting the application to app stores, I'm happy for you to use bits and pieces of code as you see fit in your own applications, but you can't submit an exact copy of one of the example applications to the app store.

## Generate Icons and Splash Screens

There's so many different devices out there with different screen sizes and resolutions, and we need to cater to all of these when creating our applications icon and splash screen.

Nobody in their right mind would be excited by the chance to make 50 different versions of the same file. Luckily there's a very easy, almost completely automated, option provided by Ionic. We can use the `ionic resources` command to generate these for us. Although Ionic does a lot for you but it can't quite yet design your graphical resources for you, so there's a little set up work we need to do.

### Design your icon

The Ionic resources tool only requires a single 192 x 192 icon as a base, but since the app store requires a larger icon anyway it's best to design it at 1024 x 1024 first (or even 2048 x 2048). For the most part, scaling down large images into small icons works well, but if you want a really crisp looking icon it may be worth designing a smaller version of your icon specifically.

**> Create a 192 x 192 icon called `icon.png` and save it inside of the `resources` folder (overwriting the existing icon)**

**Design your splash screen**

Icons are great because they are always square. Splash screens unfortunately come in all sorts of shapes and sizes. This means that you're either going to have to make your design flexible, or make different designs for each individual splash screen size.

What's a "flexible" design? Well, that's just what I've decided to call a design that will play nicely with Ionic's resource tool. If you design the splash screen at 2208 x 2208 then all of your splash screens should all automatically generate nicely.

The problem is though that there is always going to be a degree of cropping that occurs - some screens are tall and some screens are wide. To avoid important parts of your design getting chopped off you should make sure to place the important parts of the design near the center of the image, and the bits near the edges should be unimportant (e.g just part of a back- ground image).

**> Grab the splash-screen-template.psd or splash-screen-template.png file from your download pack. Create your own 2208 x 2208 splash screen based on this template and save it as `splash.png` in the resources folder along with the icon.png file you created previously.**

**Run the resources tool**

Now that you have the base icon and splash screen created, you can create all of the rest of the splash screens and icons by running the following command:

```
ionic resources
```

## Set the Bundle ID and App Name

An important step before building is to set your App Name and the Bundle ID in your **config.xml** file. If you take a look at the first few lines of this file you will see this:

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="io.ionic.starter" version="0.0.1"
   xmlns="http://www.w3.org/ns/widgets"
   xmlns:cdv="http://cordova.apache.org/ns/1.0">
    <name>V2 Test</name>
    <description>An Ionic Framework and Cordova project.</description>
    <author email="hi@ionicframework" href="http://ionicframework.com/">Ionic
       Framework Team</author>
```

You should update these details to reflect your own, including the **id** which is currently **io.ionic.starter**. This will need to match the values you use to sign your application in the next lessons, and should be set to something like **com.yourname.yourproject**. You will likely also want to set the **version** to **1.0.0** when you are releasing your application for the first time.

## Set Cordova Preferences

This is not an essential step but you can also provide some preferences in your **config.xml** file which will affect how it is built. There are some defaults included in the **config.xml** file like this:

```
    <preference name="webviewbounce" value="false" />
    <preference name="UIWebViewBounce" value="false" />
    <preference name="DisallowOverscroll" value="true" />
    <preference name="android-minSdkVersion" value="16" />
    <preference name="BackupWebStorage" value="none" />
```

But there are a ton more preferences you can specify, take a look here for more information.

## Minify Assets

This is also an optional step but an important one I think. You want to keep your final build sizes down as much as possible and one way to help do that is to minify all of your images using a tool like TinyPNG. Just run your images through the tool and save them back into your project.

## Signing iOS Applications on a Mac or PC

iOS and Android applications need to be ‘signed’. The idea is similar to a normal signature, in that it proves you authorize the action being taken and that you have the authority to do so. Signing works slightly differently for both platforms.

iOS applications will require a **.p12** file which is made up of a key and either a development or distribution certificate. As well as the **.p12** file, you will also need to create a provisioning profile. The purpose of the **.p12** file is to identify you as an iOS developer, and the provisioning profile identifies the application, services and devices that are allowed to install the application. It is quite a process to get all these things in order, but if you just follow this guide step by step you shouldn’t have too many problems.

If you have a Mac you can use XCode to handle some of this process automatically, but I will be going through everything step by step so that you can use alternate methods to build if you do not have a Mac.

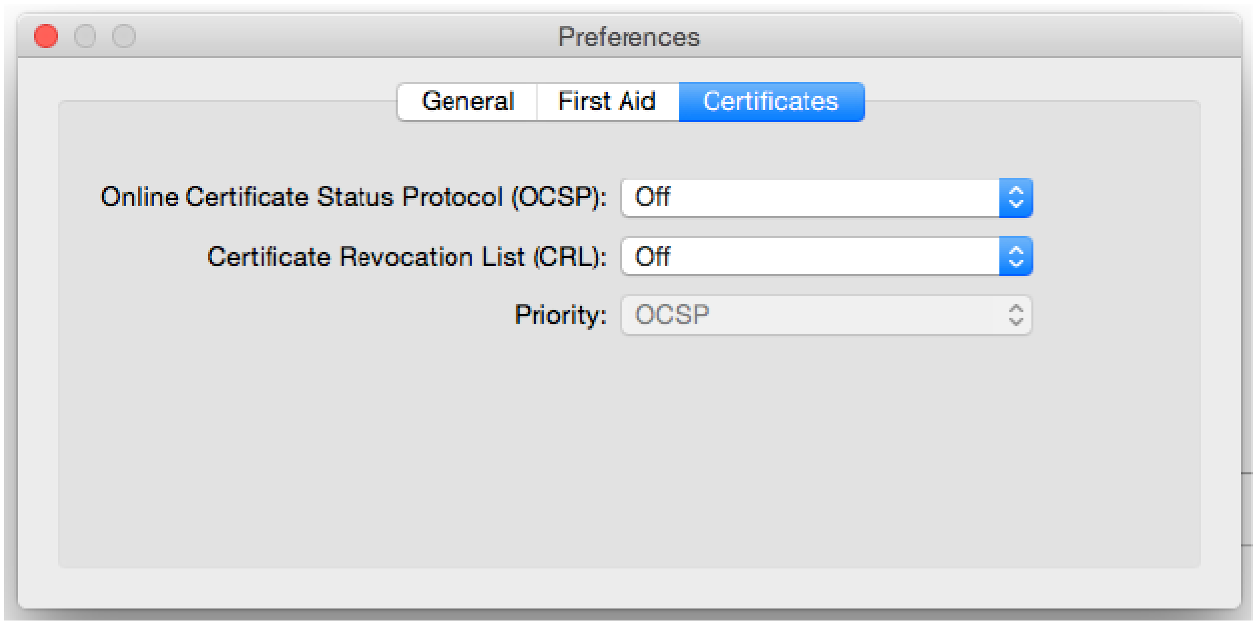
Before you get started you will need to be enrolled in the iOS Developer Program.

### Signing iOS Applications on a Mac

If you have a Mac you should follow these steps, if not skip this section and start reading **Signing iOS Applications on Windows**. You should make sure you have XCode installed before you begin.

#### Generate a Certificate Signing Request

- Open **Keychain Access**
- Go to **Keychain Access > Preferences:**


Preferences

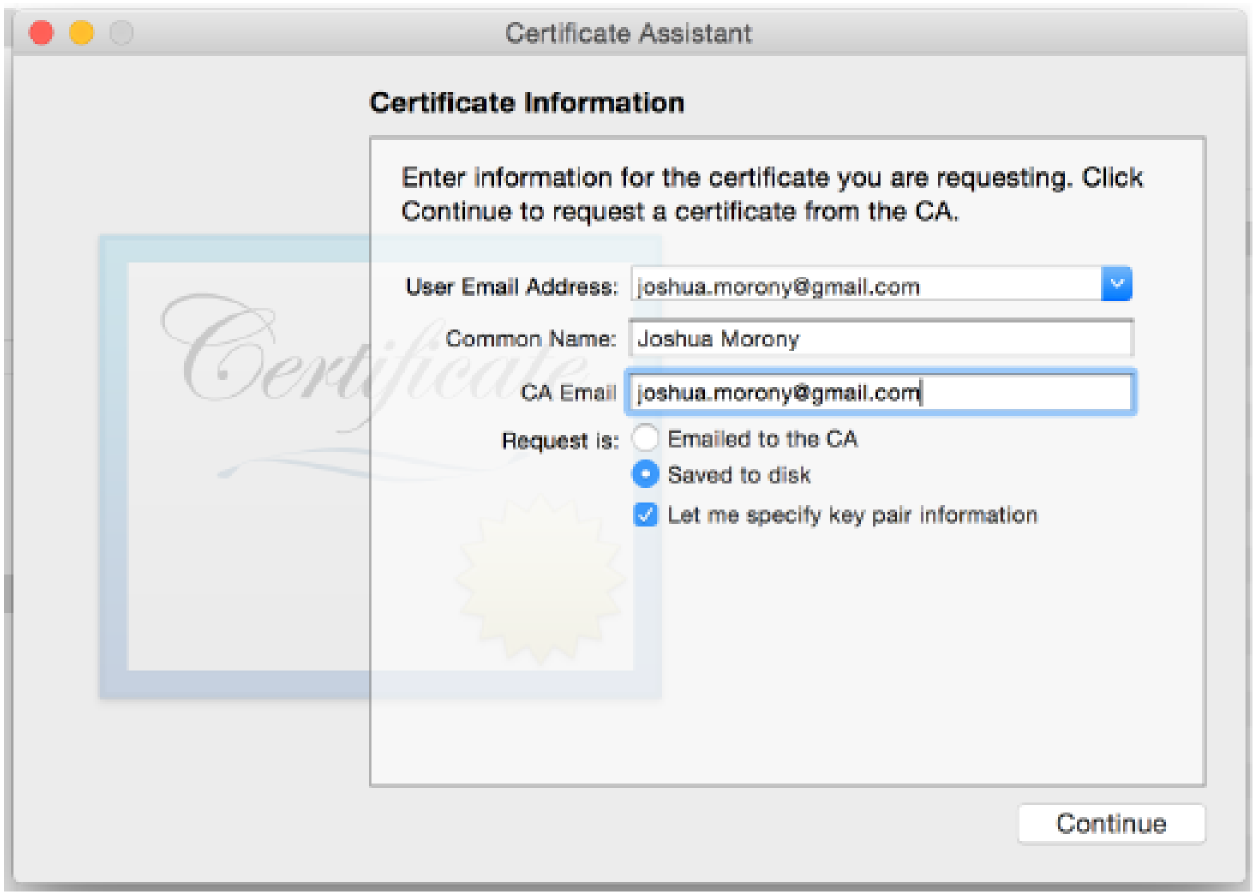
General First Aid Certificates

Online Certificate Status Protocol (OCSP): Off

Certificate Revocation List (CRL): Off

Priority: OCSP

* Go to the **Certificates** tab and make sure both options are switched off * Now go to **Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority** * Fill in the details in this window and select **Saved to disk** and **Let me specify key pair information**:


Certificate Assistant

**Certificate Information**

Enter information for the certificate you are requesting. Click Continue to request a certificate from the CA.

User Email Address: joshua.morony@gmail.com

Common Name: Joshua Morony

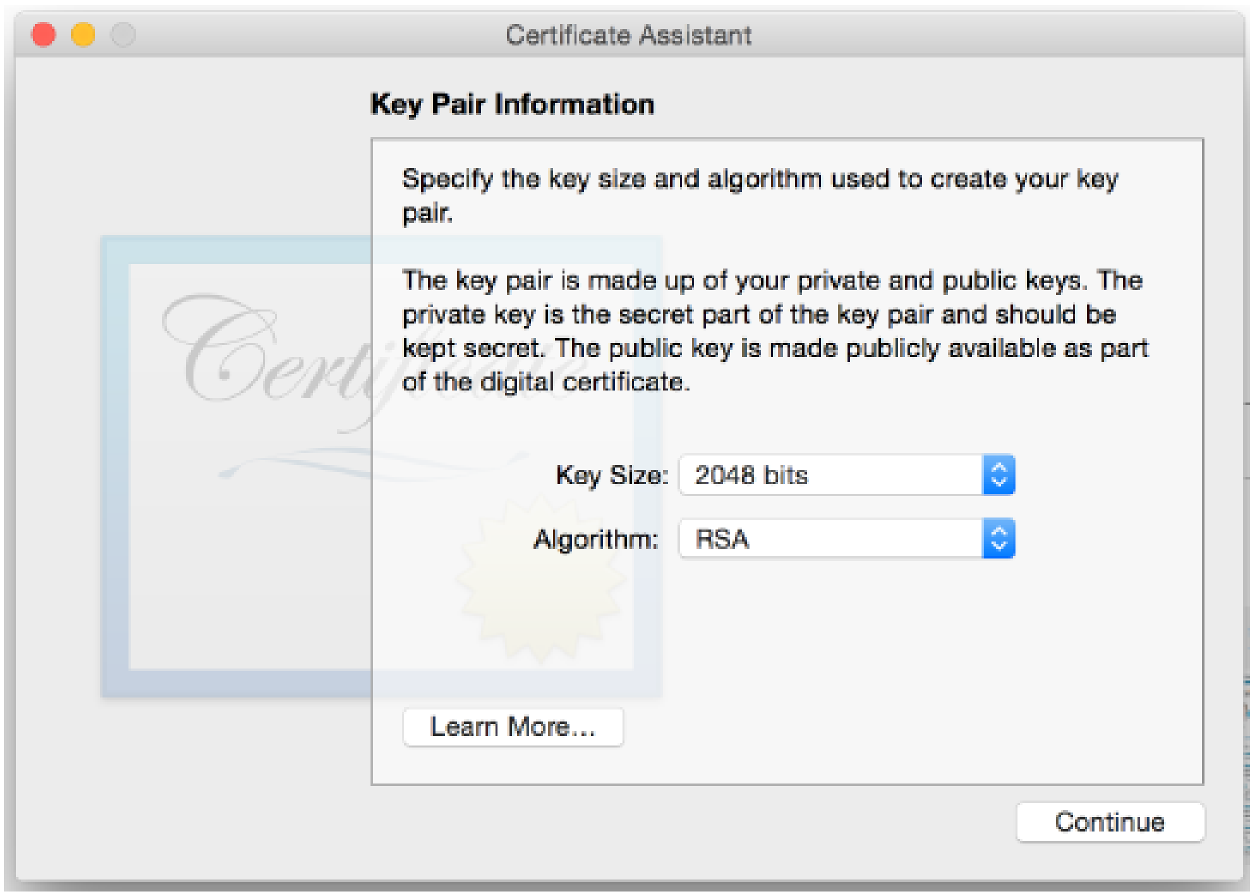
CA Email: joshua.morony@gmail.com

Request is: Emailed to the CA [✓ Saved to disk]

[✓ Let me specify key pair information]

Continue

*Click continue and choose a location to save the file on your machine* On the next screen choose **2048 bits** and **RSA**:



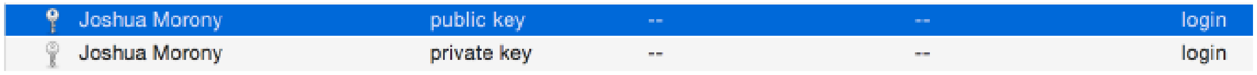
Specify the key size and algorithm used to create your key pair.

The key pair is made up of your private and public keys. The private key is the secret part of the key pair and should be kept secret. The public key is made publicly available as part of the digital certificate.

Key Size: 2048 bits

Algorithm: RSA

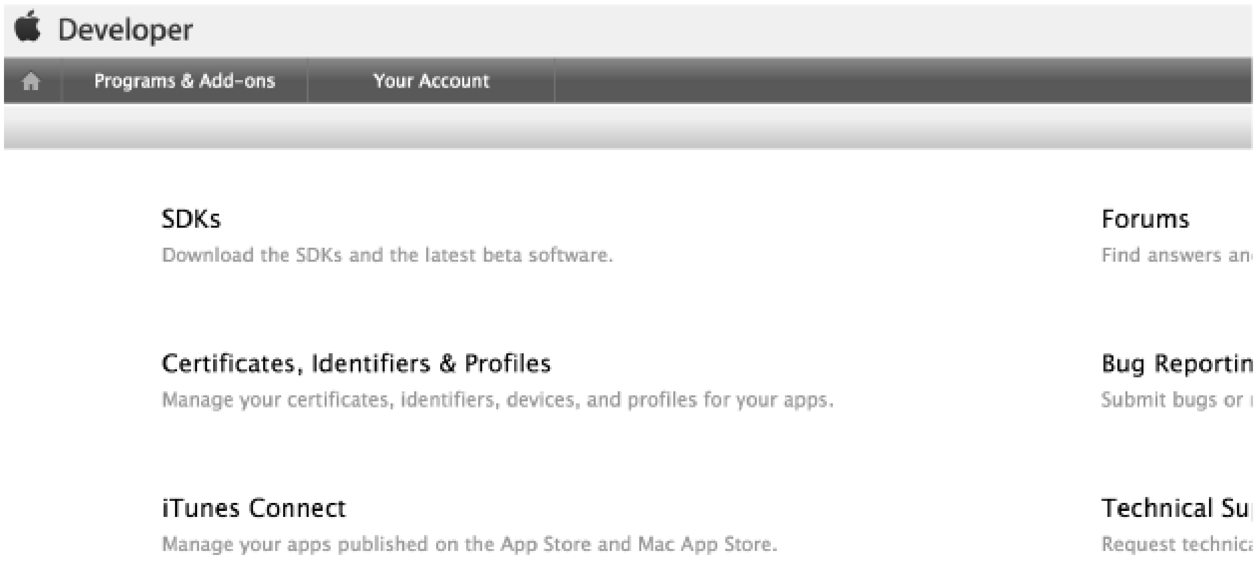
The certificate signing request will now be saved in the folder you specified and you will also notice that a public and private key pair will be created in Keychain Access:



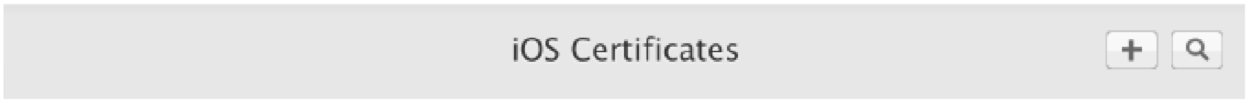
| | | | | |
|---|---|---|---|---|
| Joshua Morony | public key | -- | -- | login |
| Joshua Morony | private key | -- | -- | login |

## Create a Certificate

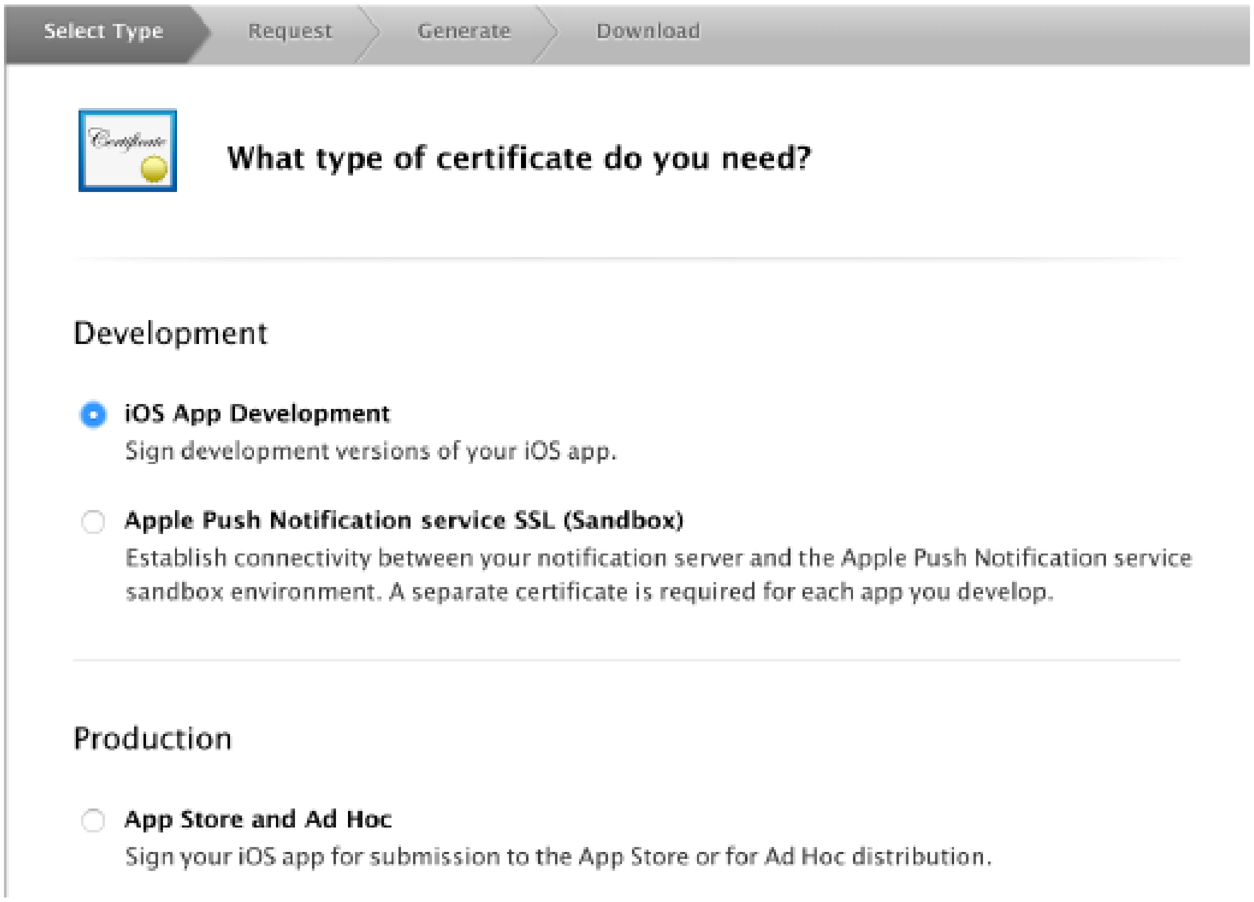
To complete these next steps you will need to log in to the Apple Member Center and go to the **Certificates, Identifiers & Profiles** section:


Developer
Programs & Add-ons
Your Account
SDKs
Download the SDKs and the latest beta software.
Forums
Find answers an
Certificates, Identifiers & Profiles
Manage your certificates, identifiers, devices, and profiles for your apps.
Bug Reportin
Submit bugs or
iTunes Connect
Manage your apps published on the App Store and Mac App Store.
Technical Su
Request technic

* 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**.



What type of certificate do you need?

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.

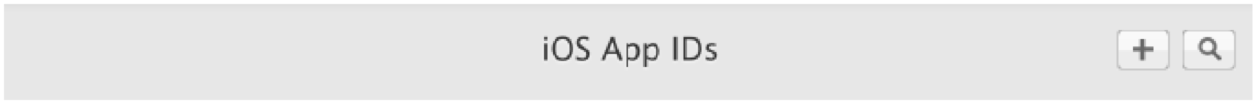
- Select your certificate type and click **Continue**
- It will now ask you for the certificate signing request you just created with Keychain Access, click continue and then upload the signing request. Once you have selected the **.certSigningRequest** file click **Generate**.

You will now be able to download your certificate. Download it to somewhere safe and then open it to install it.

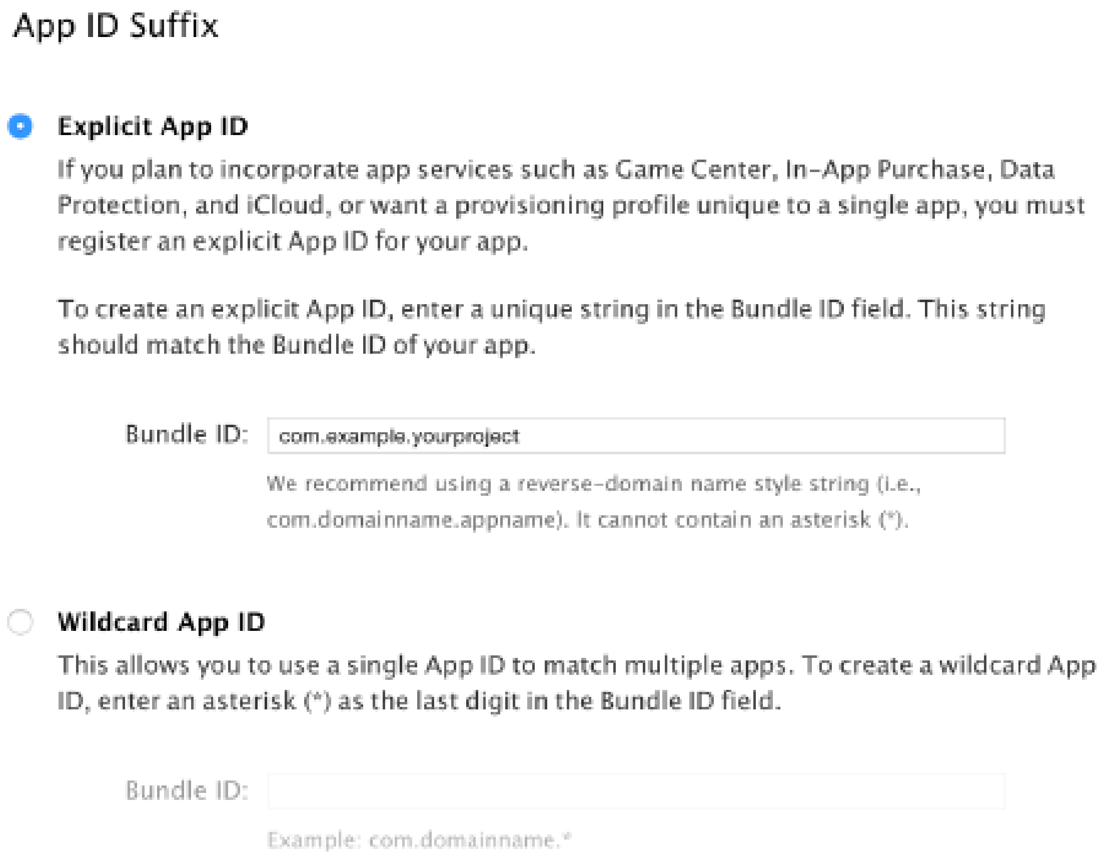
**Create an Identifier**

If you're managing your app using XCode then you don't necessarily need to worry about these step as XCode can use a Wildcard App ID, but you can also create your own App ID manually (the Bundle ID created here will match what is in your config.xml file). Here's how you do it:

- 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: com.example.yourproject

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.

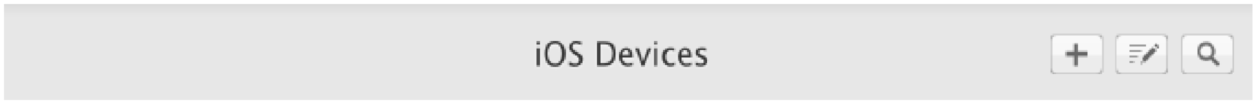
- Go to the bottom of the screen and click **Continue**
- On the following screen click **Submit**

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.

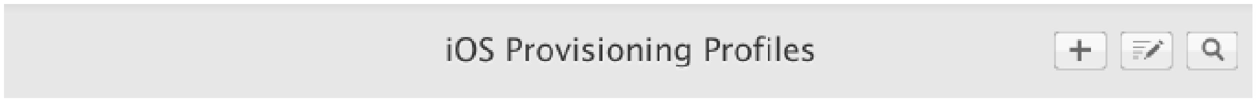
- Click on **Devices** and then click the **+** button in the top right:



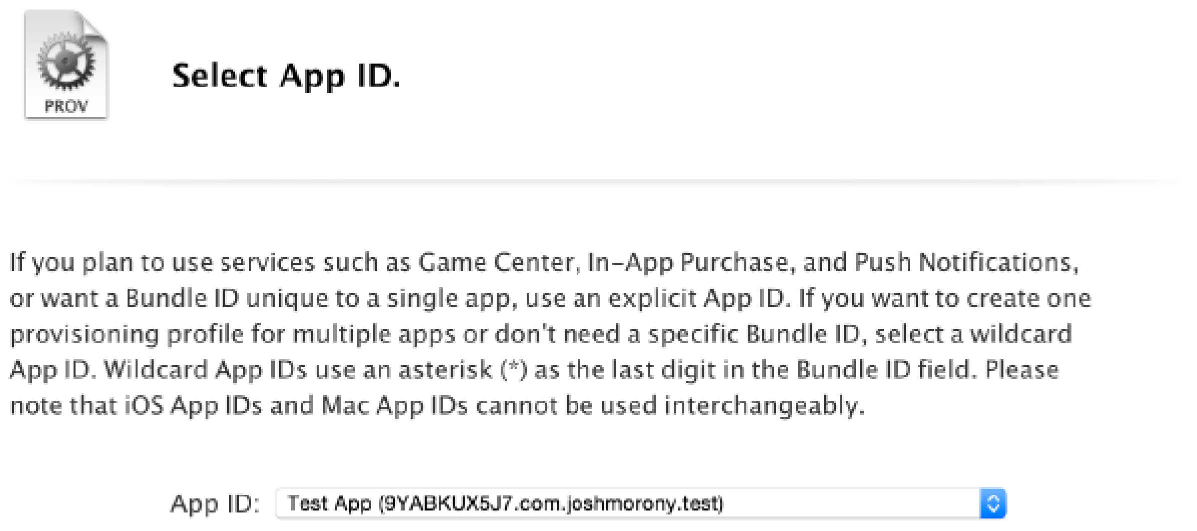
* Supply the name and UDID (you can find this for your device in iTunes) of the device you want to register and then click **Continue**. Follow the prompts to finish.

Now we can create the provisioning profile.

*Go to the provisioning profiles screen and click the **+** icon in the top right



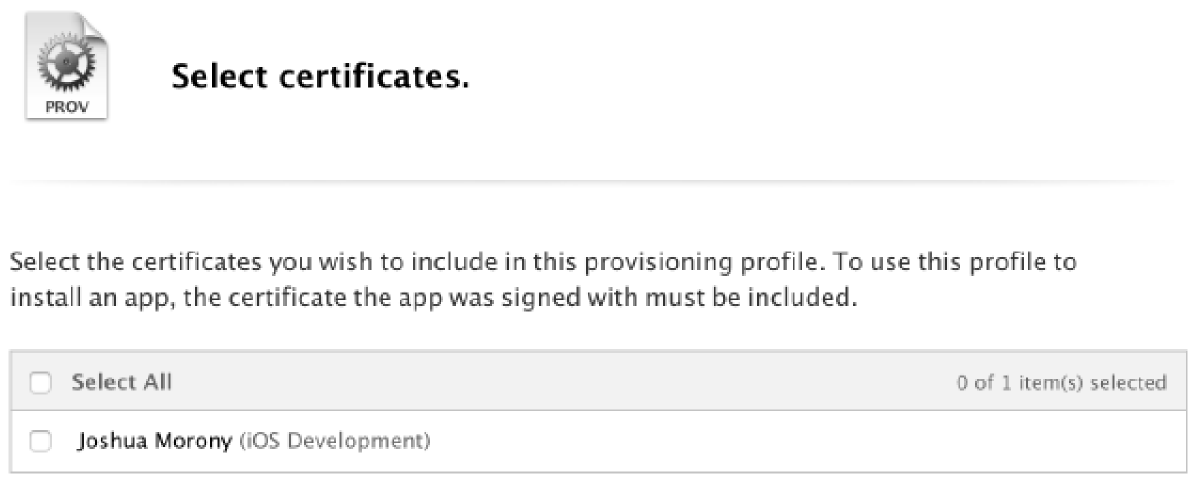
* Choose **iOS App Development** if you are creating a provisioning profile for testing, or **App Store** if you are creating a provisioning profile for distribution. Click **Continue**. * Select the App ID you just created and click **Continue**:


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: Test App (9YABKUX5J7.com.joshmorony.test)

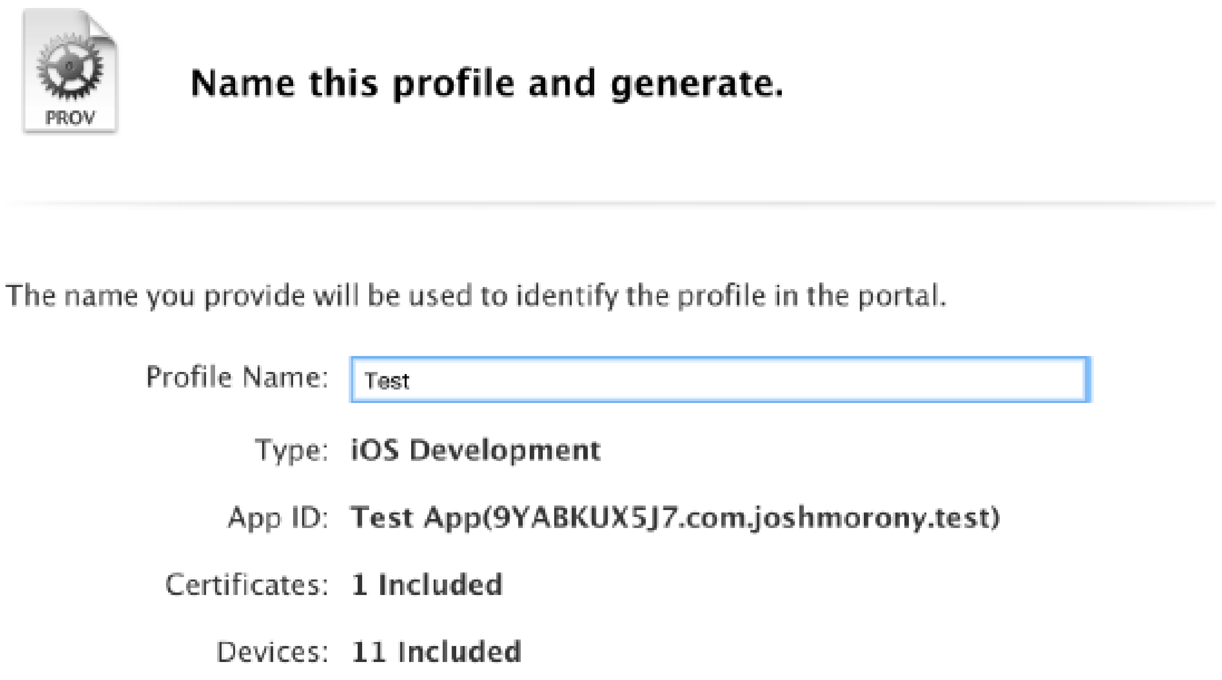
* Select the certificate you wish to use and click **Continue**:


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.

|  |  |
|---|---|
| ☐ Select All | 0 of 1 item(s) selected |
| ☐ 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: Test

Type: iOS Development

App ID: Test App(9YABKUX5J7.com.joshmorony.test)

Certificates: 1 Included

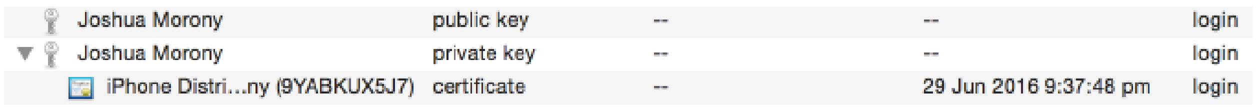
Devices: 11 Included

You should now be able to download your Provisioning Profile.

**Generating a .p12**

A **.p12 file** (along with the provisioning profile) is required along with the provisioning profile for services like PhoneGap Build. Since you have a Mac this step is not really relevant to you, however you can still use a service like PhoneGap Build if you would like (I wouldn't recommend it). If you do need to create a .p12 file on your Mac, here's how you do it:

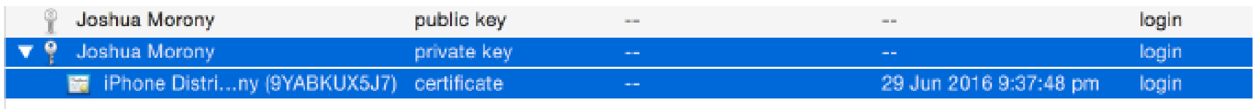
- Open Keychain Access
- Back in the first step when we created the certificate signing request, I mentioned that a public and private key pair would be generated in Keychain Access. Find this in the **Keys** section:



| | | | | |
|---|---|---|---|---|
| Joshua Morony | public key | -- | -- | login |
| Joshua Morony | private key | -- | -- | login |
| iPhone Distri...ny (9YABKUX5J7) | certificate | -- | 29 Jun 2016 9:37:48 pm | login |

You will see now that the private key also has the certificate you created in the iOS Member Center attached to it.

- Highlight both the private key and the certificate, right click and choose **Export 2 items…**:



| | | | | |
|---|---|---|---|---|
| Joshua Morony | public key | -- | -- | login |
| Joshua Morony | private key | -- | -- | login |
| iPhone Distri...ny (9YABKUX5J7) | certificate | -- | 29 Jun 2016 9:37:48 pm | login |

* You will then be asked where you would like to save the file and you may choose to enter a password for the **.p12** file or leave it blank. You may also be asked to enter in the Admin password for your computer.

Congrats! You now have your **.p12** file.

## Signing iOS Applications on Windows

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
```

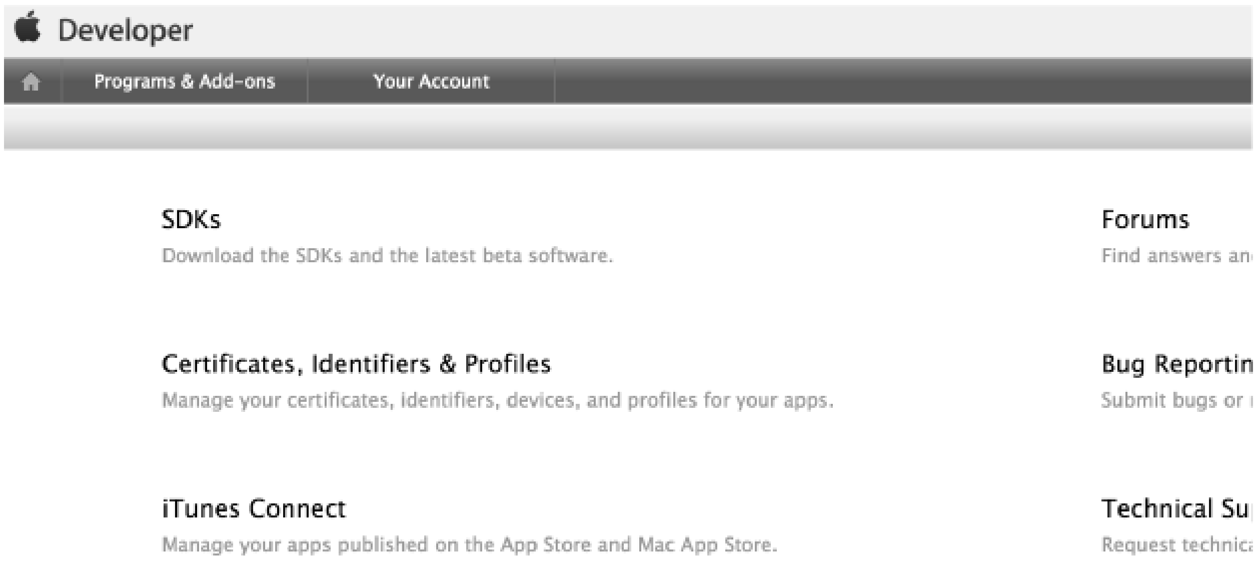
- Use that key to generate a certificate signing request

```
openssl req -new -key mykey.key -out myCSR.certSigningRequest -subj
   "/emailAddress=you@yourdomain.com, CN=Your Name, C=AU"
```

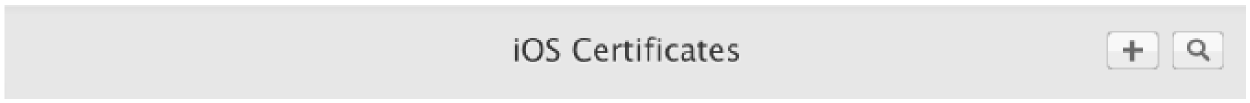
Make sure to replace the email address, name and country code with your own.

**Create a Certificate**

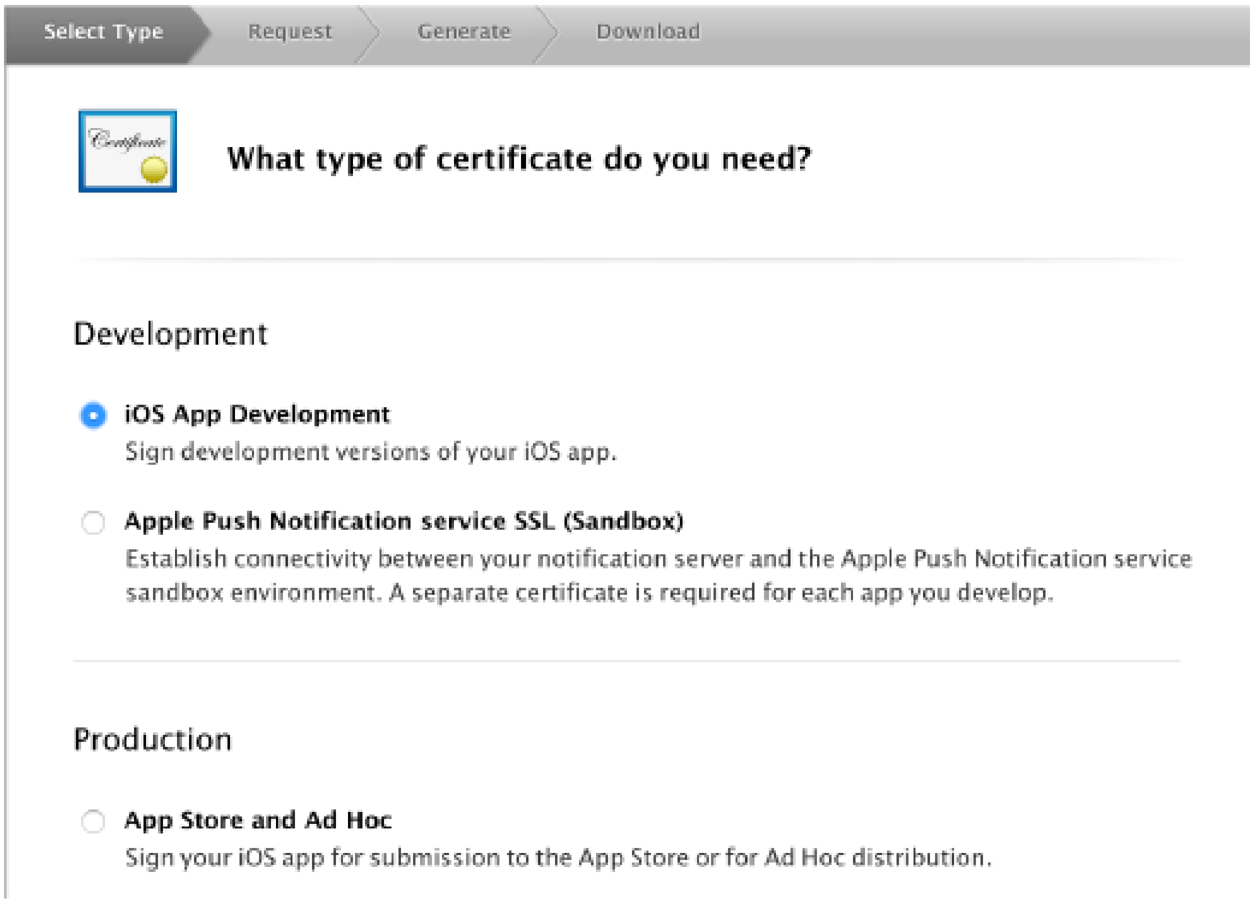
To complete these next steps you will need to log in to the Apple Member Center and go to the **Certificates, Identifiers & Profiles** section:


Developer
Programs & Add-ons
Your Account
SDKs
Download the SDKs and the latest beta software.
Forums
Find answers an
Certificates, Identifiers & Profiles
Manage your certificates, identifiers, devices, and profiles for your apps.
Bug Reportin
Submit bugs or
iTunes Connect
Manage your apps published on the App Store and Mac App Store.
Technical Su
Request technic

* 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**.


Select Type Request Generate Download

What type of certificate do you need?

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.

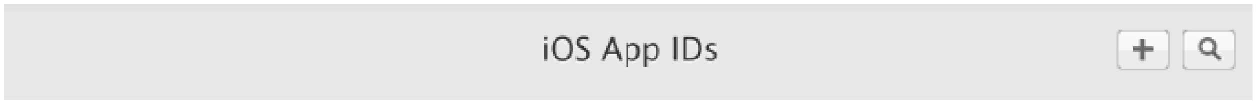
You will also notice some other options on this screen.  You will need to create separate certificates if you want to use things like the Apple Push Notifications service or WatchKit, but we won't require any of those.

- 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**.

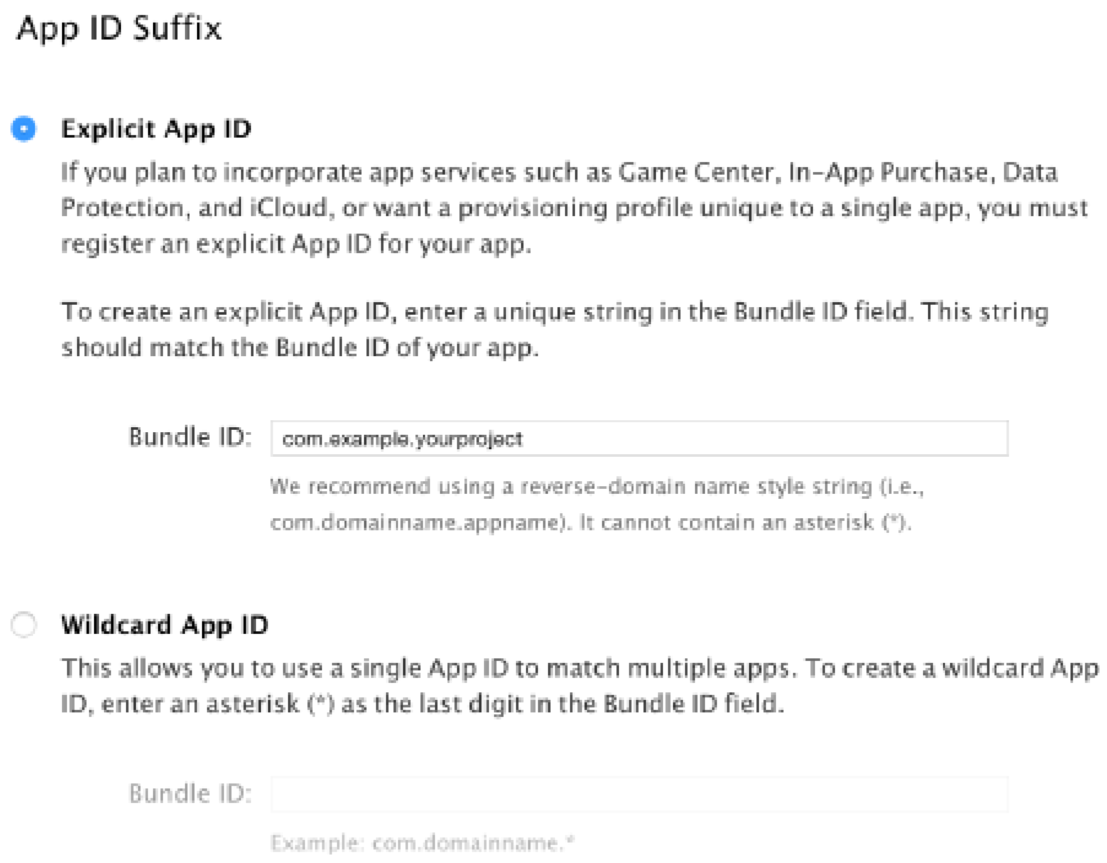
You will now be able to download your certificate.  Download it to somewhere safe and then open it to install it (for convenience, you should place it in your OpenSSL bin folder (e.g.  **OpenSSL-Win64/bin** because this is where we will be running some commands later).

**Create an Indentifier**

- 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: com.example.yourproject

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.

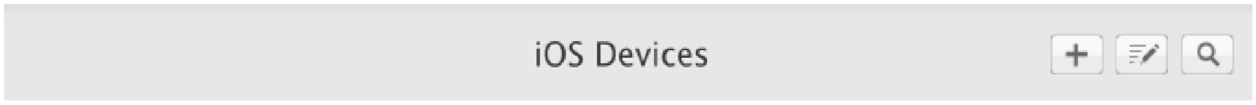
- Go to the bottom of the screen and click **Continue**
- On the following screen click **Submit**

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.

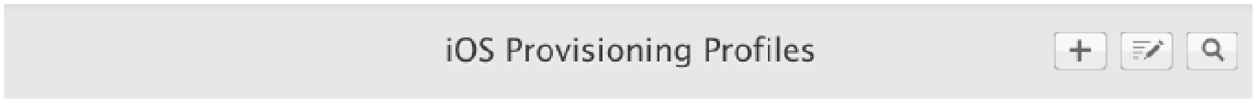
- Click on **Devices** and then click the **+** button in the top right:



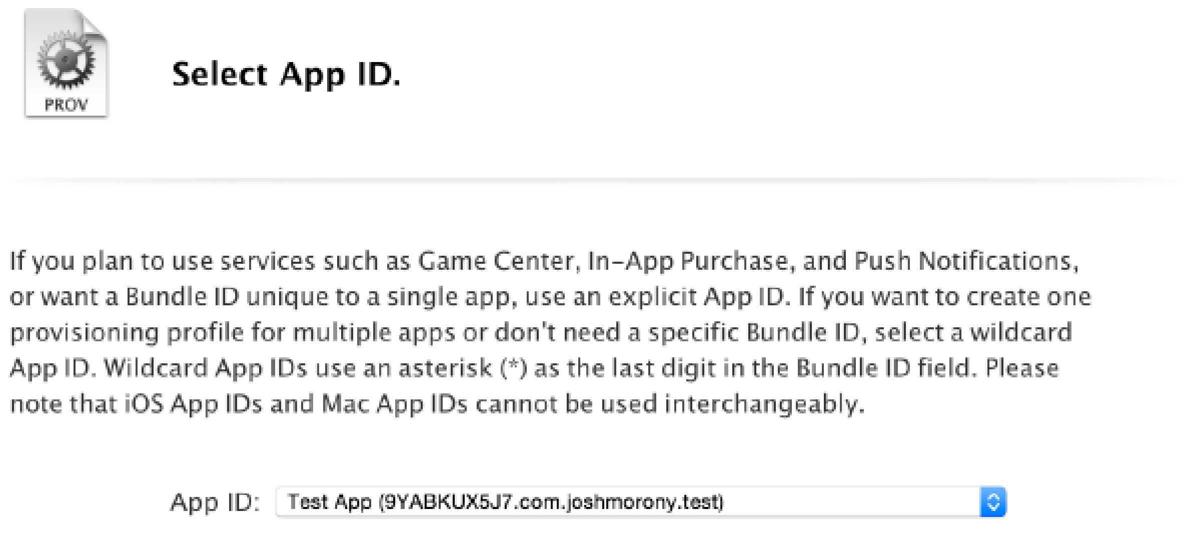
* Supply the name and UDID (you can find this for your device in iTunes) of the device you want to register and then click **Continue**. Follow the prompts to finish.

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**:

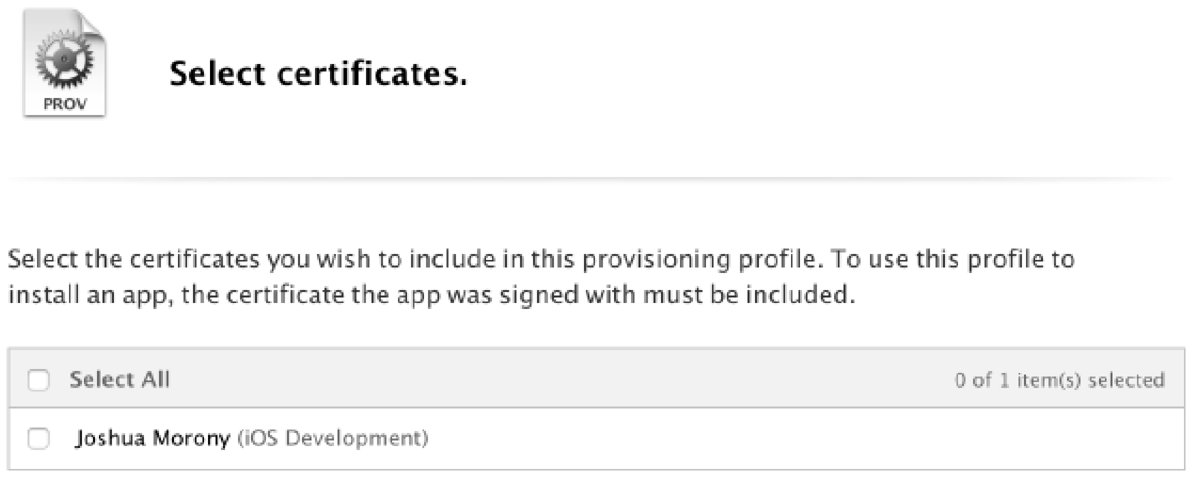


Select App ID.

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: Test App (9YABKUX5J7.com.joshmorony.test)

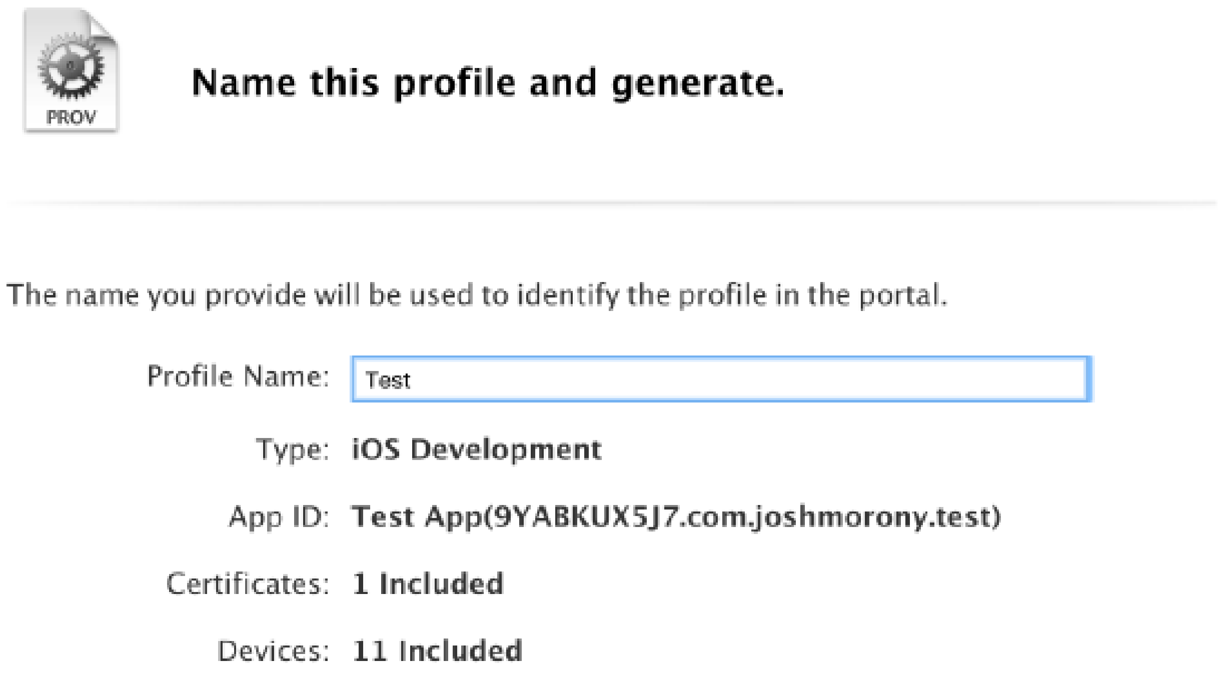
* Select the certificate you wish to use and click **Continue**:


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.

| | |
|---|---|
| ☐ Select All | 0 of 1 item(s) selected |
| ☐ 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: | Test |
| 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's 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 PhoneGap Build 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!).

We will discuss exactly how to build with PhoneGap Build shortly.

## Signing Android Applications on a Mac or PC

As I mentioned before, Android is a little different to iOS (and a bit easier fortunately). iOS requires that you create certificates and provisioning profiles for both development and production applications.

Android however, will let you install a ‘debug’ application on a testing device without having to go through this process. You will only need to sign your application when you want to release it on Google Play. Android requires that you create a ‘keystore’ file to sign your application with.

### Signing an Android Application

It is considerably easier to sign an Android application and, as I mentioned before, you don’t even need to sign the application if you just want to test it (only if you are releasing it to Google Play). To sign our Android application we will need to create a **keystore** file.

To sign applications we will be using a tool called **keytool** that comes with the Android SDK. If you do not have the Android SDK set up on your machine already you can follow this guide:

http://ionicframework.com/docs/ionic-cli-faq/#android-sdk

Once you have the Android SDK set up, you can follow these steps to create a keystore file.

- Run the following command to generate a keystore file with an alias of **alias_name** (you should change this)

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name
-keyalg RSA -keysize 2048 -validity 10000
```

When you enter this command you will be prompted for a password - choose one and make sure to store or remember it. You will then be asked a series of questions, most of which you can leave blank if you wish. Finally, you will be asked for a keystore password.

You will now have a keystore file that you can use to sign your application (we will discuss how that works later).

**IMPORTANT:** In order to update your application on the app store later you will need both the keystore file, along with the alias name and password. If you lose any of these you will not be able to update the application.

## Generating a Key Hash

If you are using Facebook functionality that requires creating a Facebook application which you would have done if you completed the CamperChat application in this book (only available in the Expert package) then you will need to create a Key Hash from your keystore file to supply to Facebook for Android.

Doing that is simple enough, all you need to do is 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 have done this your Key Hash will be output to the terminal, and then you can simply copy it over to the Android platform settings in your Facebook application.

## Building for iOS & Android Using PhoneGap Build (without MAC)

There are two ways you can incorporate Cordova / PhoneGap into your applications, you can either use a locally installed version through the command line or you can use the PhoneGap Build cloud service.

The choice should be simple:

- If you have a Mac: use Cordova locally
- If you're not building an iOS application: use Cordova locally
- If you do not have a Mac and want to create iOS applications: use PhoneGap Build

The PhoneGap Build service will allow you to create 1 private application for free. You can keep deleting and reusing this allowance to build multiple applications if you like, but if you ever want to manage multiple applications in PhoneGap Build at the same time then you will need to purchase a subscription.

**IMPORTANT: Unless you fit into that 3rd option (No Mac but want to build for iOS) you should SKIP THIS LESSON**

PhoneGap Build is a great service, but you should always use a local installation of Cordova where possible. The advantages of using Cordova locally are:

- Build times are quicker since you don't need to the PhoneGap Build server and then download the resulting .ipa (and .apk for Android) files
- You don't need to use your data to download and upload the files and you won't require an Internet connection for development
- You have access to a wider range of plugins
- You have more control over the application (you have direct access to the native wrapper, rather than just the web files)
- PhoneGap Build requires payment if you want more than 1 private repository

I did mention that PhoneGap Build can be used to build iOS applications without a Mac. It can also be used to create Android applications without the Android SDK. This is why it doesn't matter what operating system you are using, since all of the compilation happens on PhoneGap Build's servers.

When using PhoneGap Build you are able to upload your application code (purely web based code, i.e: HTML, CSS and JavaScript) and have compiled application files returned for iOS and Android (as well as for Windows Phone). The general process looks something like this:

1. Build your application
2. Create a config.xml file that tells PhoneGap Build how your application should be built
3. Zip up your application
4. Upload to PhoneGap Build
5. Download the resulting native packages

Compilation and the integration of SDK's are handled on their end, which makes this approach much easier initially than getting everything set up correctly on your machine (but harder in the long run). Most importantly **it makes it possible to create iOS applications without a Mac**.

By this point you should have an application finished and ready to build, so let's walk through how to do that with PhoneGap Build.

## Building with PhoneGap Build

When uploading to PhoneGap Build we only want to include the built web files, that is everything inside of the **www** folder in your project. All the rest of the files and folders are mainly for configuring Cordova and the build process which we don't need when using PhoneGap Build.

There's a couple of little changes we need to make before uploading to PhoneGap Build.

### Create a config.xml file

As you may already be aware, your project contains a **config.xml** file that is used for configuring how your application is built with Cordova. We still need this file, but the one required by PhoneGap Build is a bit different.

**WARNING:** The next code block is huge and hard to read in this format, so if you prefer you can just copy the code from the example PhoneGap Build application that came with your download pack.

**> Create a file at `www/config.xml` and add the following**

```
<?xml version="1.0" encoding="UTF-8" ?>
    <!-- Change the id to something like com.yourname.yourproject -->
    <widget xmlns   = "http://www.w3.org/ns/widgets"
        xmlns:gap   = "http://phonegap.com/ns/1.0"
        id          = "com.example.project"
        versionCode = "10"
        version     = "1.0" >

    <!-- versionCode is optional and Android only -->

    <name>App Name</name>

    <description>
        Description
    </description>

    <author href="http://www.example.com" email="you@yourdomain.com">
        Your Name
    </author>

    <plugin name="cordova-sqlite-storage" source="npm" />
    <plugin name="com.phonegap.plugin.statusbar" source="pgb" />
    <plugin name="org.apache.cordova.splashscreen" source="pgb" />
    <plugin name="com.indigoway.cordova.whitelist.whitelistplugin"
       source="pgb" />
```

```
<preference name="prerendered-icon" value="true" />
<preference name="target-device" value="universal" />
<preference name="android-windowSoftInputMode" value="stateAlwaysHidden"
   />

<icon src="resources/android/icon/drawable-ldpi-icon.png"
   gap:platform="android" gap:qualifier="ldpi"/>
<icon src="resources/android/icon/drawable-mdpi-icon.png"
   gap:platform="android" gap:qualifier="mdpi"/>
<icon src="resources/android/icon/drawable-hdpi-icon.png"
   gap:platform="android" gap:qualifier="hdpi"/>
<icon src="resources/android/icon/drawable-xhdpi-icon.png"
   gap:platform="android" gap:qualifier="xhdpi"/>
<icon src="resources/android/icon/drawable-xxhdpi-icon.png"
   gap:platform="android" gap:qualifier="xxhdpi"/>
<icon src="resources/android/icon/drawable-xxxhdpi-icon.png"
   gap:platform="android" gap:qualifier="xxxhdpi"/>

<icon src="resources/ios/icon/icon.png" gap:platform="ios" width="57"
   height="57"/>
<icon src="resources/ios/icon/icon@2x.png" gap:platform="ios" width="114"
   height="114"/>
<icon src="resources/ios/icon/icon-40.png" gap:platform="ios" width="40"
   height="40"/>
<icon src="resources/ios/icon/icon-40@2x.png" gap:platform="ios"
   width="80" height="80"/>
<icon src="resources/ios/icon/icon-50.png" gap:platform="ios" width="50"
   height="50"/>
<icon src="resources/ios/icon/icon-50@2x.png" gap:platform="ios"
   width="100" height="100"/>
```

```
<icon src="resources/ios/icon/icon-60.png" gap:platform="ios" width="60"
   height="60"/>
<icon src="resources/ios/icon/icon-60@2x.png" gap:platform="ios"
   width="120" height="120"/>
<icon src="resources/ios/icon/icon-60@3x.png" gap:platform="ios"
   width="180" height="180"/>
<icon src="resources/ios/icon/icon-72.png" gap:platform="ios" width="72"
   height="72"/>
<icon src="resources/ios/icon/icon-72@2x.png" gap:platform="ios"
   width="144" height="144"/>
<icon src="resources/ios/icon/icon-76.png" gap:platform="ios" width="76"
   height="76"/>
<icon src="resources/ios/icon/icon-76@2x.png" gap:platform="ios"
   width="152" height="152"/>
<icon src="resources/ios/icon/icon-small.png" gap:platform="ios"
   width="29" height="29"/>
<icon src="resources/ios/icon/icon-small@2x.png" gap:platform="ios"
   width="58" height="58"/>

<gap:splash src="resources/android/splash/drawable-land-ldpi-screen.png"
   gap:platform="android" gap:qualifier="land-ldpi"/>
<gap:splash src="resources/android/splash/drawable-land-mdpi-screen.png"
   gap:platform="android" gap:qualifier="land-mdpi"/>
<gap:splash src="resources/android/splash/drawable-land-hdpi-screen.png"
   gap:platform="android" gap:qualifier="land-hdpi"/>
<gap:splash src="resources/android/splash/drawable-land-xhdpi-screen.png"
   gap:platform="android" gap:qualifier="land-xhdpi"/>
<gap:splash
   src="resources/android/splash/drawable-land-xxhdpi-screen.png"
   gap:platform="android" gap:qualifier="land-xxhdpi"/>
```

```
<gap:splash
   src="resources/android/splash/drawable-land-xxxhdpi-screen.png"
   gap:platform="android" gap:qualifier="land-xxxhdpi"/>
<gap:splash src="resources/android/splash/drawable-port-ldpi-screen.png"
   gap:platform="android" gap:qualifier="port-ldpi"/>
<gap:splash src="resources/android/splash/drawable-port-mdpi-screen.png"
   gap:platform="android" gap:qualifier="port-mdpi"/>
<gap:splash src="resources/android/splash/drawable-port-hdpi-screen.png"
   gap:platform="android" gap:qualifier="port-hdpi"/>
<gap:splash src="resources/android/splash/drawable-port-xhdpi-screen.png"
   gap:platform="android" gap:qualifier="port-xhdpi"/>
<gap:splash
   src="resources/android/splash/drawable-port-xxhdpi-screen.png"
   gap:platform="android" gap:qualifier="port-xxhdpi"/>
<gap:splash
   src="resources/android/splash/drawable-port-xxxhdpi-screen.png"
   gap:platform="android" gap:qualifier="port-xxxhdpi"/>

<gap:splash src="resources/ios/splash/Default-568h@2x~iphone.png"
   gap:platform="ios" width="640" height="1136"/>
<gap:splash src="resources/ios/splash/Default-667h.png" width="750"
   gap:platform="ios" height="1334"/>
<gap:splash src="resources/ios/splash/Default-736h.png" width="1242"
   gap:platform="ios" height="2208"/>
<gap:splash src="resources/ios/splash/Default-Landscape-736h.png"
   gap:platform="ios" width="2208" height="1242"/>
<gap:splash src="resources/ios/splash/Default-Landscape@2x~ipad.png"
   gap:platform="ios" width="2048" height="1536"/>
<gap:splash src="resources/ios/splash/Default-Landscape~ipad.png"
   gap:platform="ios" width="1024" height="768"/>
```

```
    <gap:splash src="resources/ios/splash/Default-Portrait@2x~ipad.png"
       gap:platform="ios" width="1536" height="2048"/>
    <gap:splash src="resources/ios/splash/Default-Portrait~ipad.png"
       gap:platform="ios" width="768" height="1024"/>
    <gap:splash src="resources/ios/splash/Default@2x~iphone.png"
       gap:platform="ios" width="640" height="960"/>
    <gap:splash src="resources/ios/splash/Default~iphone.png"
       gap:platform="ios" width="320" height="480"/>

    <!-- Default Icon and Splash -->
    <icon src="resources/icon.png" />
    <gap:splash src="resources/splash.png" />

    <access origin="*"/>

</widget>
```

This file is specific for the Quicklists application - notice how we've got a few lines that include plugins? The way plugins are included with PhoneGap Build is different to Cordova. You may remember at the start of building each application we run a bunch of commands like this:

```
ionic plugin add [plugin name]
```

to set up the plugins. This sets up the plugin locally in the project, but to use a plugin with PhoneGap Build we need to specific it in our new **config.xml** file like this:

```
    <plugin name="cordova-sqlite-storage" source="npm" />
    <plugin name="com.phonegap.plugin.statusbar" source="pgb" />
    <plugin name="org.apache.cordova.splashscreen" source="pgb" />
    <plugin name="com.indigoway.cordova.whitelist.whitelistplugin"
       source="pgb" />
```

Although most are, not all plugins are available for PhoneGap Build. When adding the plugins for a project, just search for the PhoneGap Build installation instructions, most plugins will include information on what you need to include for PhoneGap Build. So if you are building any of the applications in this book with PhoneGap Build, you should look in the **Getting Ready** section and make sure to include any plugins we are using in this **config.xml** file.

Also make sure you remember to replace the **id** at the top with your own Bundle ID, i.e: `com.yourproject.yourname`.

**Copy the Resources**

The other main difference is the way the resources (splash screens and icons) are included. We also need to add references to these to our **config.xml** file and we need to copy over our resources because currently they live outside of the **www** folder (so they wouldn't be included in our upload)

> Copy the **resources** folder inside of the **www** folder

As well as copying over the resources, you should also save a blank file with the name `.pgbomit` inside of the resources folder. This tells PhoneGap Build that the files in this folder should only be used for the splash screens and icons, and shouldn't be made available in the application (which would take up a lot of space).

**Upload to PhoneGap Build**

Now that we've got everything ready, you just need to zip up the contents of the **www** folder (NOT the **www** folder itself) and upload it to PhoneGap Build. If you do not already have an account you will have to create one at build.phonegap.com.

Once you have an account simply create a new app and upload the .zip archive you just created and click 'Ready to Build'. The iOS build will fail initially but the Android version should succeed. This is because you don't *need* to sign an Android application during development, but you do for iOS.

When you're building your release version, or if you want to download the iOS version, you will have to upload the signing keys you created in the last chapters. This means you will need to attach the **.p12** file and the Provisioning Profile to your iOS build, and the **keystore** file to your Android build.

Once the applications have finished building, you will be able to download your **.ipa** file for iOS and a **.apk** file for Android. We will discuss how to get these on the app stores in just a moment.

## Submitting to the Apple App Store

The time has finally come to send our application off to apple! Before submitting your application make sure that you familiarise yourself with the App Store Review Guidelines - if you don't comply with these guidelines your application may be rejected.

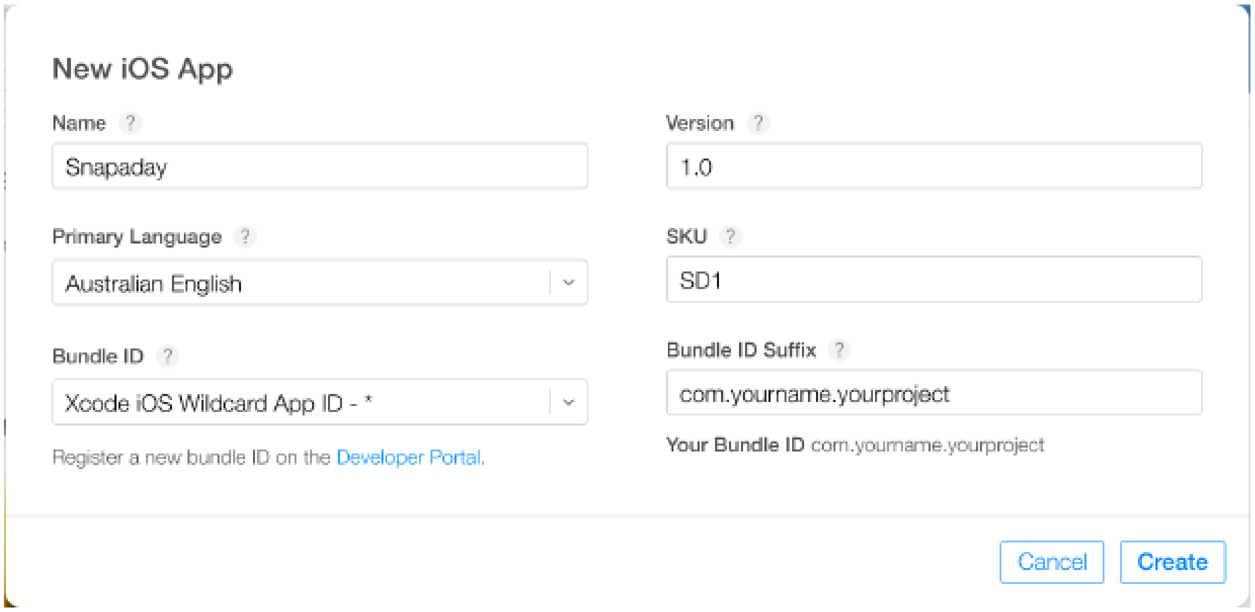
**NOTE:** Do not follow these steps until you have created your own application - You should not upload any example applications from this book.

To submit an application to the App Store you will need to create an App Store Listing and of course upload your application.

### Creating an App Store Listing

Let's start by walking through how to create an App Store Listing. There's quite a few things you need to do but it's pretty straight forward.

- Log into iTunes Connect
- Go to **My Apps**
- Select the **+** icon on the left and then choose **New iOS App**
- Fill out the details on this prompt and then click **Create**:


New iOS App

Name: Snapaday

Version: 1.0

Primary Language: Australian English

SKU: SD1

Bundle ID: Xcode iOS Wildcard App ID - *

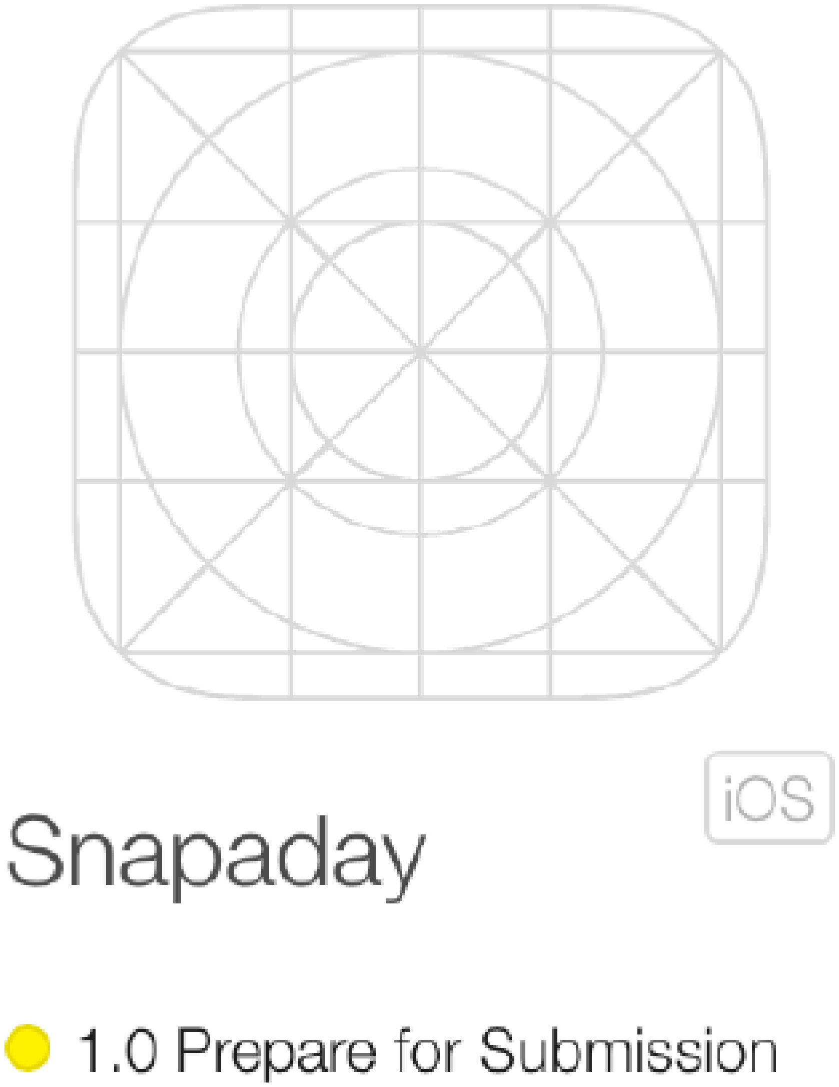
Bundle ID Suffix: com.yourname.yourproject

Register a new bundle ID on the Developer Portal.

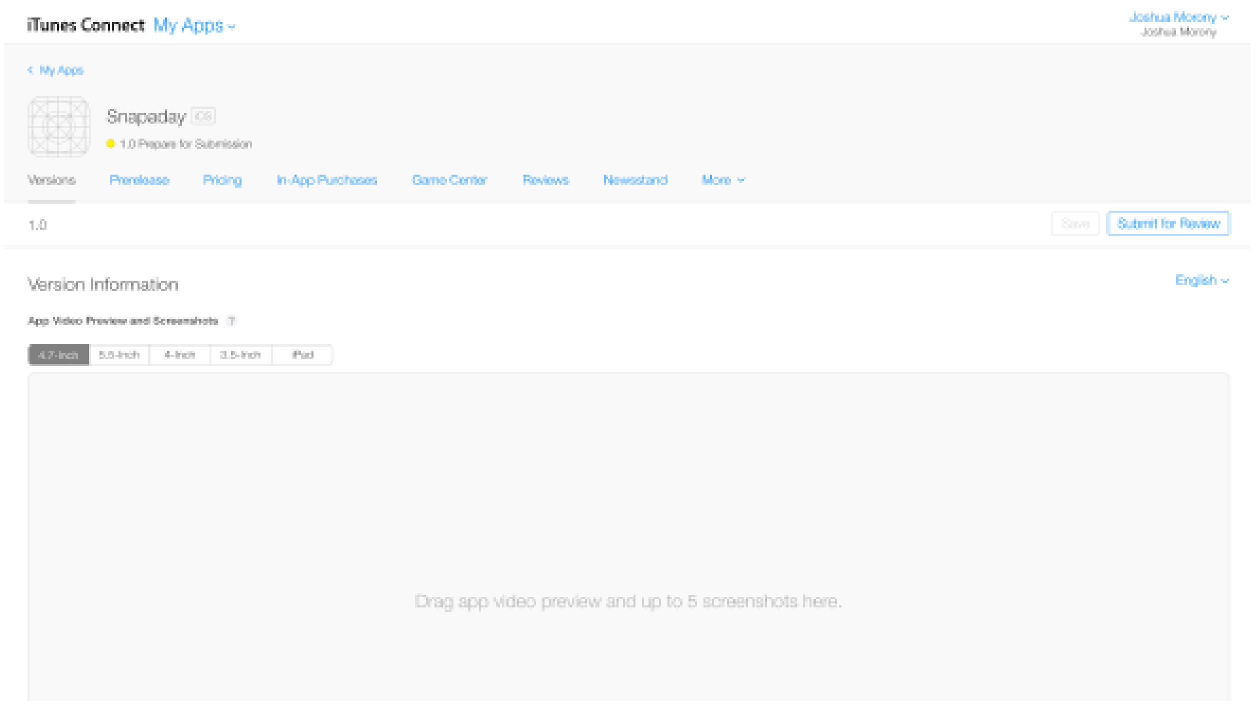
**Your Bundle ID** com.yourname.yourproject

If you're using XCode to submit your application you can use **Xcode iOS Wildcard App ID**, or otherwise you can choose the specific Bundle ID you created for your application in the iOS Certificates lesson. The SKU does not need to be anything specific, it is just for your own reference and **the Bundle ID Suffix should match the id you specified in your config.xml file**.

You should now see your application in the dashboard:



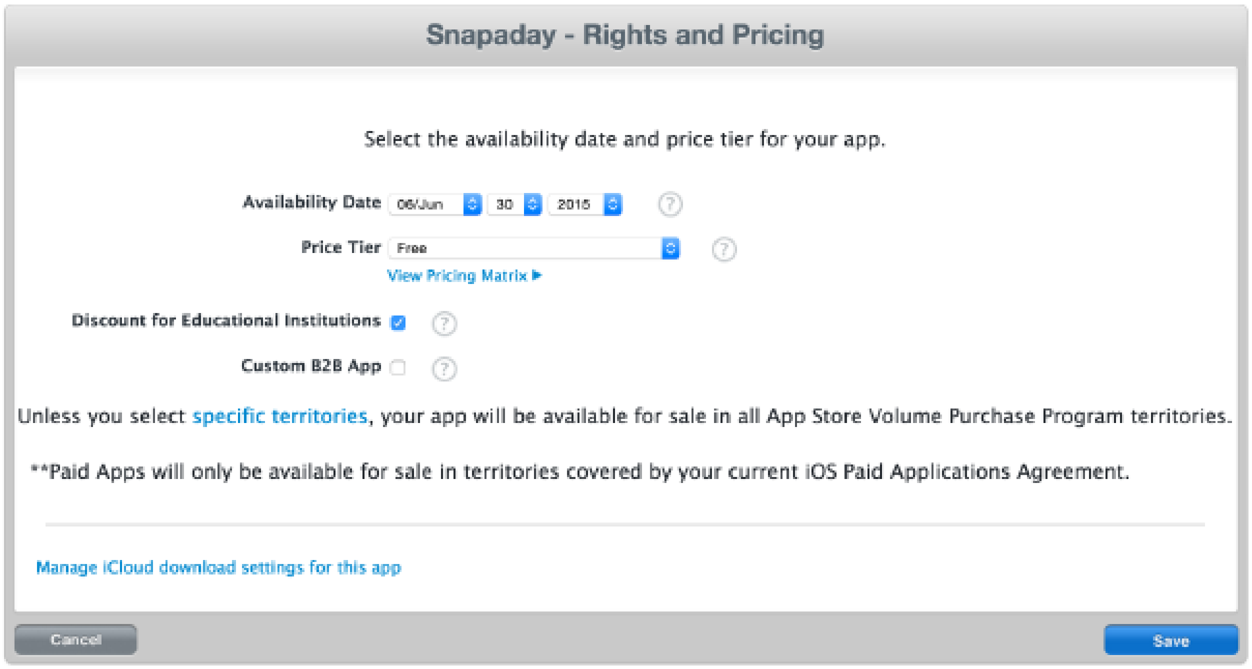
* Open your new application in iTunes Connect and you should see a dashboard like this:



* Fill out all the information on this page (including multiple screenshots for the different sized devices)

**NOTE:** You will notice a Build section on this page. You will have to come back to this section after you have uploaded your application (which we will do in the next section) and select the build you have uploaded.

- Click the **Pricing** tab and fill out the following information:



Snapaday - Rights and Pricing

Select the availability date and price tier for your app.

Availability Date: 06/Jun 30 2015

Price Tier: Free

View Pricing Matrix ▶

Discount for Educational Institutions ✓

Custom B2B App ☐

Unless you select specific territories, your app will be available for sale in all App Store Volume Purchase Program territories.

**Paid Apps will only be available for sale in territories covered by your current iOS Paid Applications Agreement.

Manage iCloud download settings for this app

If you want to release a paid application you will need to have accepted additional agreements within iTunes Connect.

## Uploading the Application

There's a few ways to do this and the way you do it will depend on what format your application is in and what operating system you are using. To submit an iOS application for distribution through the app store you will need your app signed with a distribution certificate, and again, the way you do this will depend on the method you are using.

Once you upload your application you should be able to see it in the **Build** section in iTunes Connect and will be able to attach it to your app store listing.

### Submitting an app using XCode

If you have a Mac then you can use XCode to submit the application, which is a pretty straightforward way to do it. If you do not have a Mac and instead have a .ipa file generated from using PhoneGap Build you should skip to the Submitting an app using Application Loader section.

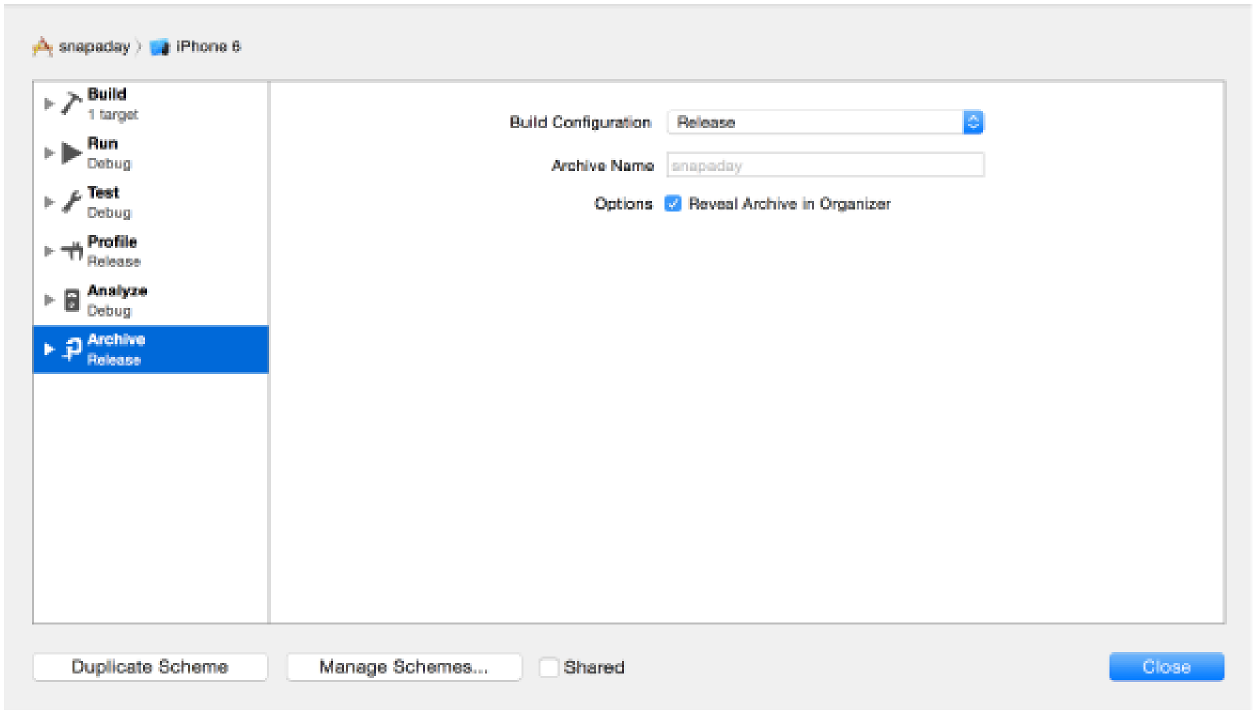
Before continuing you should run the following command within your project directory:

```
ionic build ios
```

### Converting a .xcodeproj file to .xcarchive

If you have a **.xcodeproj** file (which will be generated when you run the build command) you will first need to generate a **.xcarchive** file from it. To do that then you will need to follow these steps:

- Open your **.xcodeproj** file (located in **platforms/ios/snapaday.xcodeproj**) in XCode by double clicking it
- Go to **Product > Scheme > Edit Scheme** and make sure that the archive is set to a **Release** configuration:

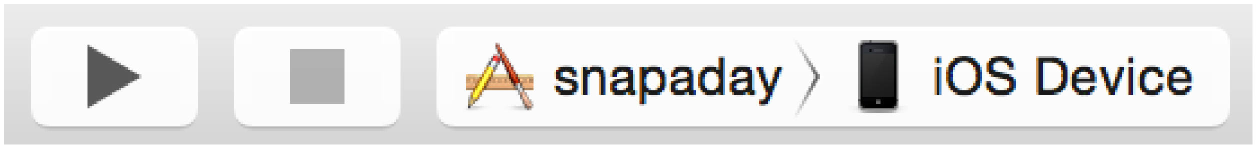

snapaday 〉 iPhone 6

Build Configuration: Release

Archive Name: snapaday

Options: ☑ Reveal Archive in Organizer

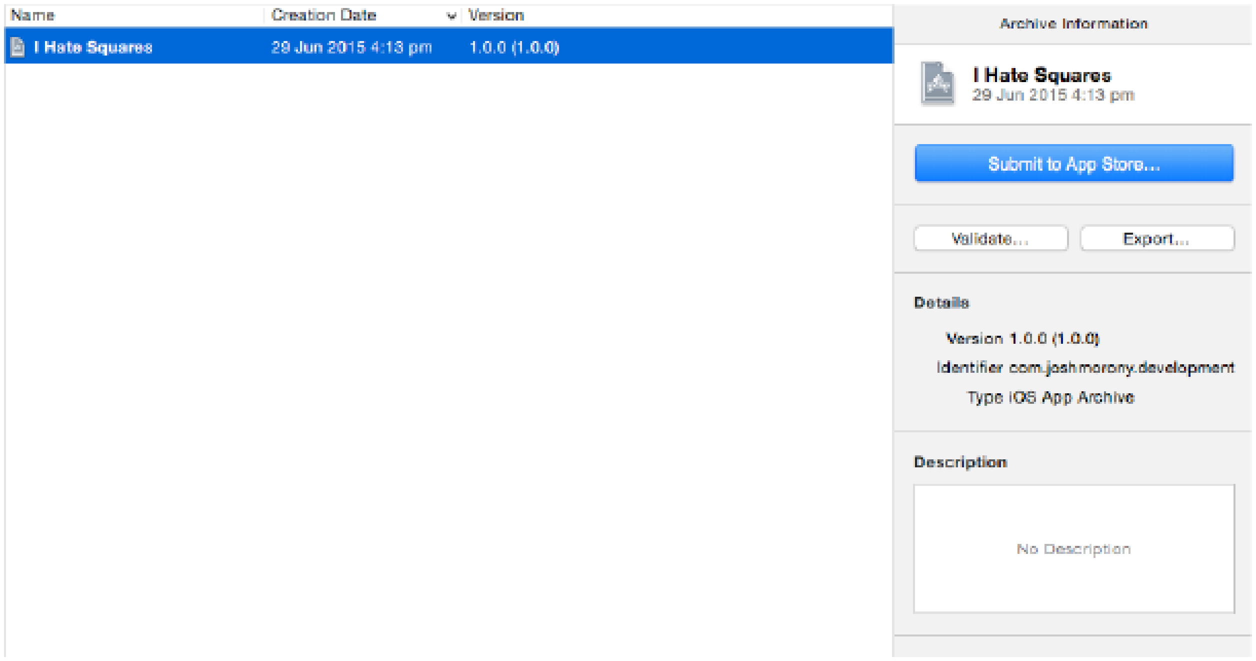
* Make sure you have **iOS Device** or **Generic iOS Device** selected in the top toolbar, not an emulator:



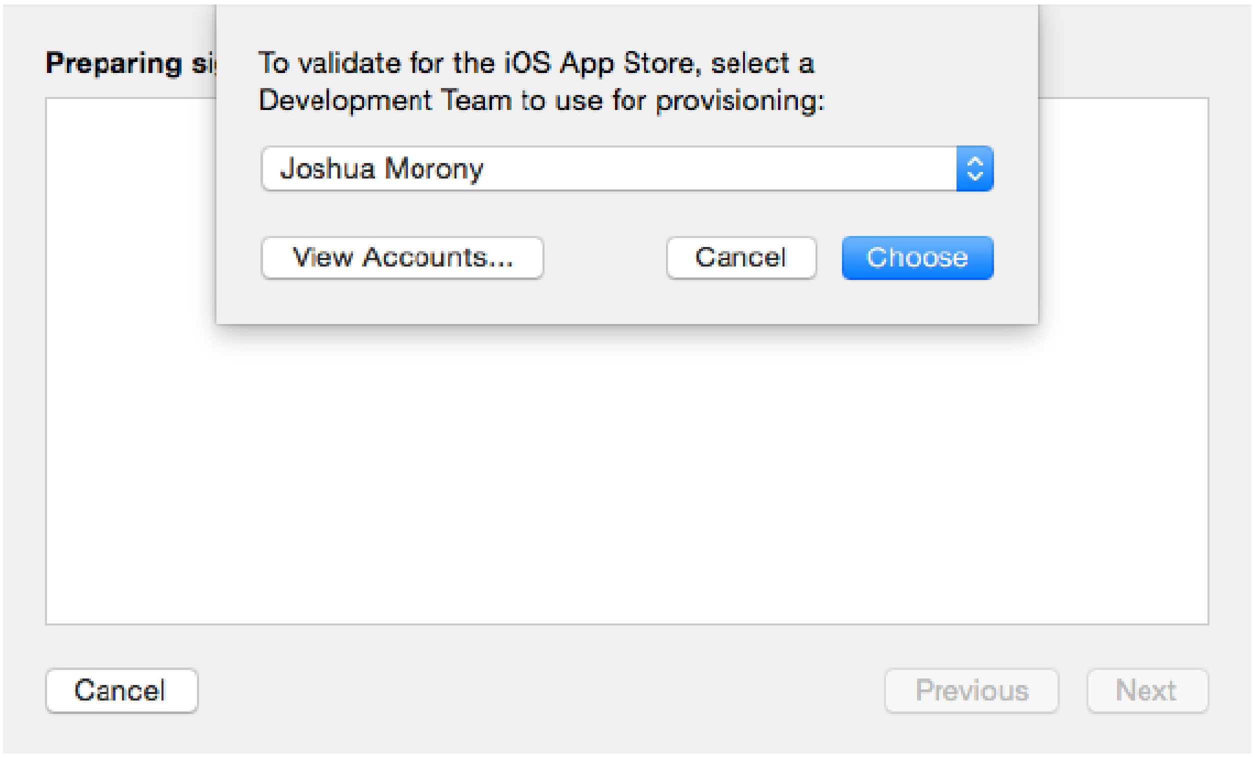
* Choose **Product > Archive**

**Uploading a .xcarchive file**

If you want to submit a **.xcarchive** first double click it to open this screen in XCode (this screen should also automatically open after you **Archive** your app):

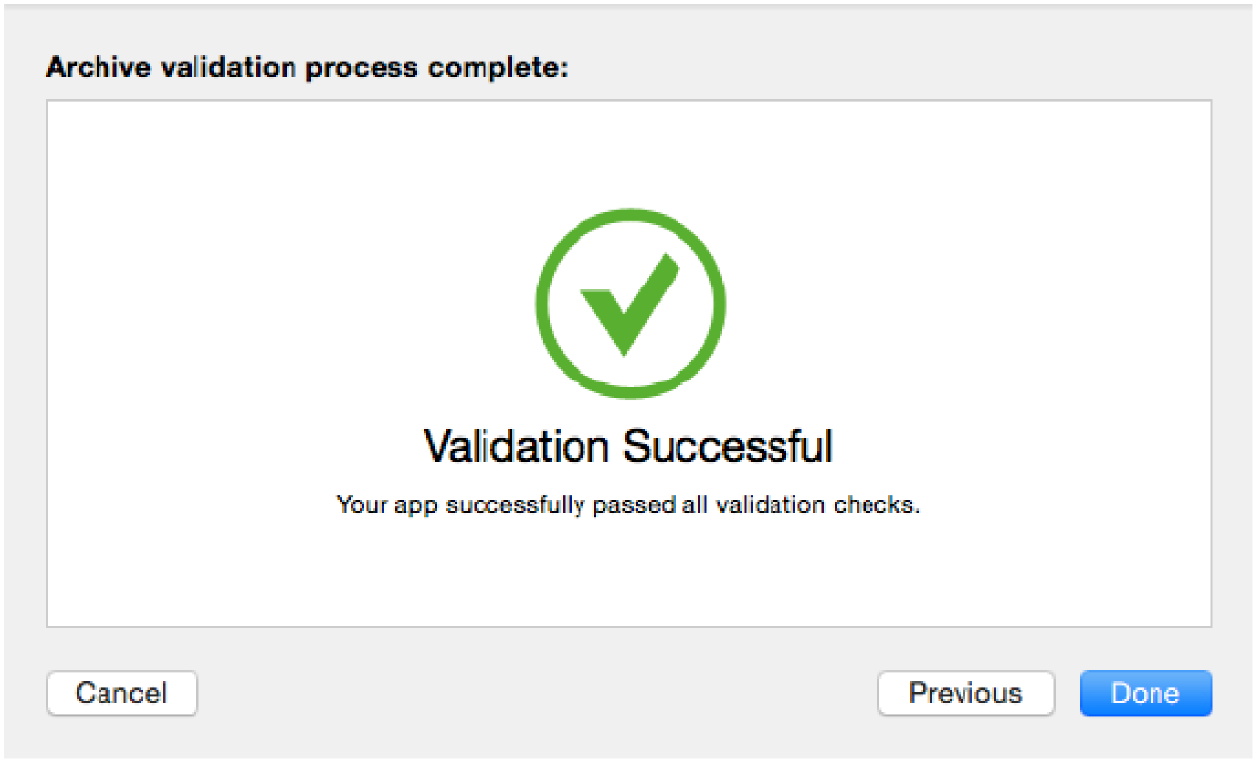


First you will want to choose your archive and then click the **Validate…** button to make sure that everything is set up correctly. You should be given the option to choose your account:



Preparing si
To validate for the iOS App Store, select a Development Team to use for provisioning:
Joshua Morony
View Accounts...
Cancel
Choose
Cancel
Previous
Next

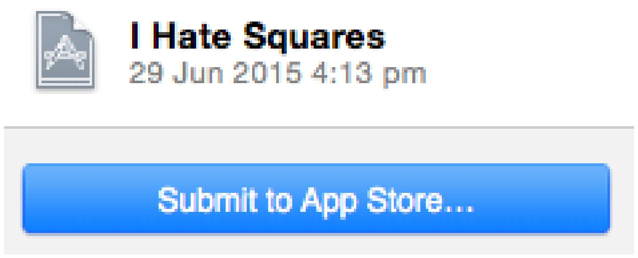
and then you will see your application displayed. Click **Validate** and if everything is set up correctly you should see this prompt:


Archive validation process complete:
Validation Successful
Your app successfully passed all validation checks.
Cancel
Previous
Done

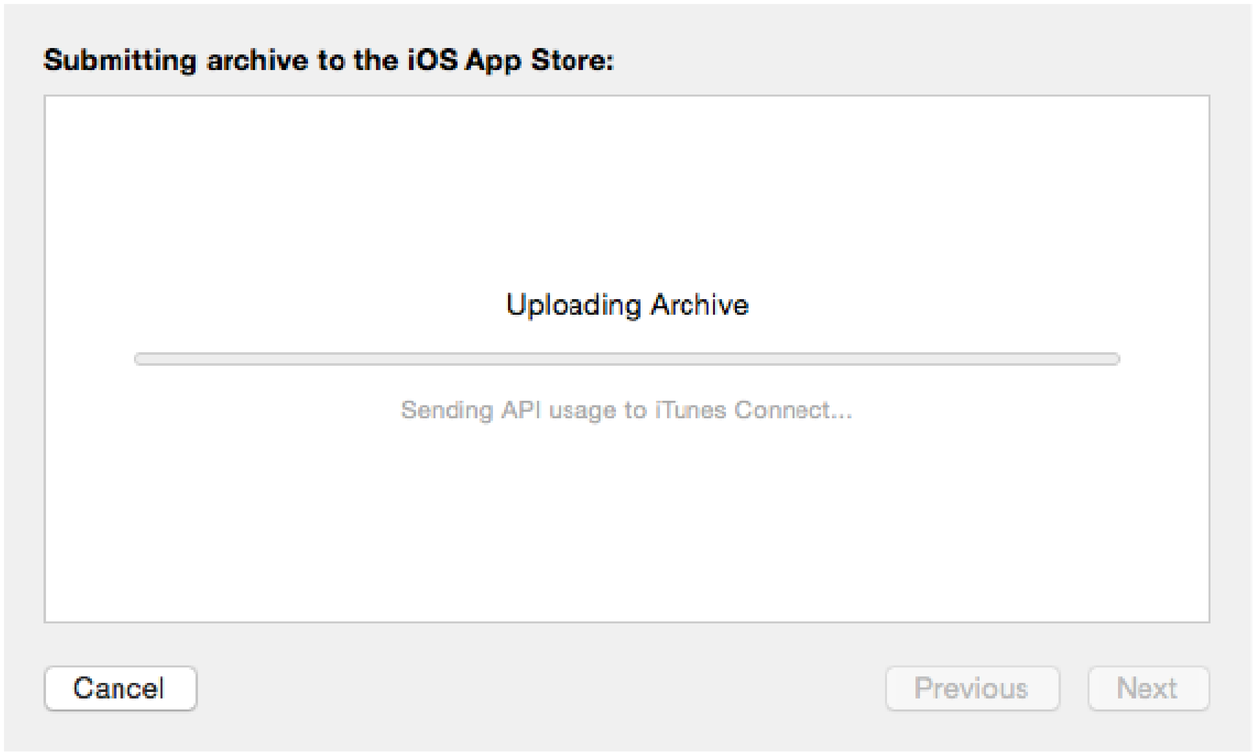
If validation does not work, make sure:

- You have set up your application in iTunes Connect
- You have followed the instructions in the iOS Certificates lesson
- The id in your config.xml matches the Bundle ID Suffix in iTunes Connect

Once you have successfully validated your project click **Done** and then choose **Submit to App Store…** or **Upload to App Store…**:



You will now go through that same process again, except this time you will choose **Submit**. Once you click **Submit** your application will begin uploading to iTunes Connect:


Submitting archive to the iOS App Store:

Uploading Archive

Sending API usage to iTunes Connect...

**Submitting an app using Application Loader**

If you do not have a Mac then your only option to submit an application is to submit the built and signed **.ipa** file. Remember, this **.ipa** file should be signed with a distribution certificate instead of a development certificate if you are submitting it to the app store.

If you do not already have a signed **.ipa** file, **make sure to read the PhoneGap Build lesson**.

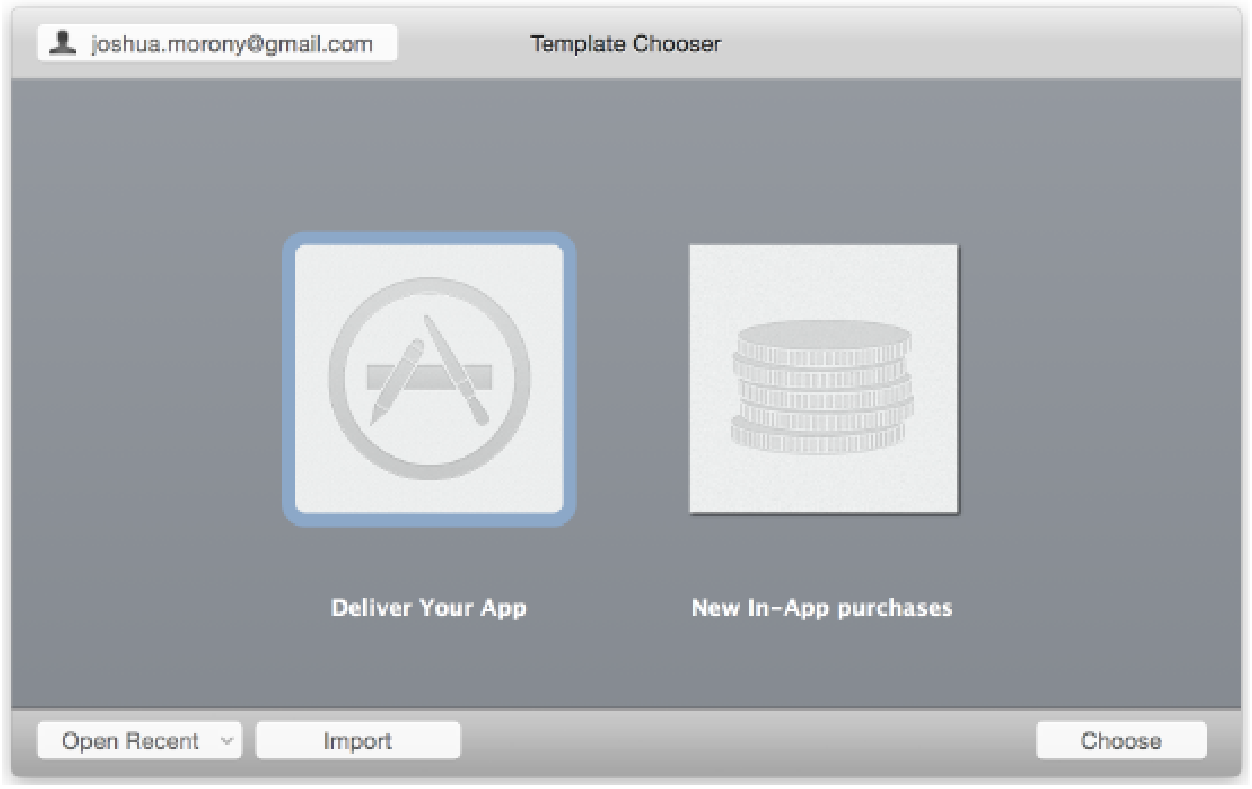
You can use a program called **Application Loader** to submit a **.ipa** file to iTunes Connect, but unfortunately this program is only available on Macs. Literally the only thing you can't do when building iOS applications on a Windows machine is upload the final file, how frustrating.

There are ways around this though, and my two favourites are:

- Borrow a friends Mac. You will only need it for about 5 minutes, so if you know anyone who has a Mac just shove your **.ipa** file on a USB stick, download Application Loader on their Mac and upload your application

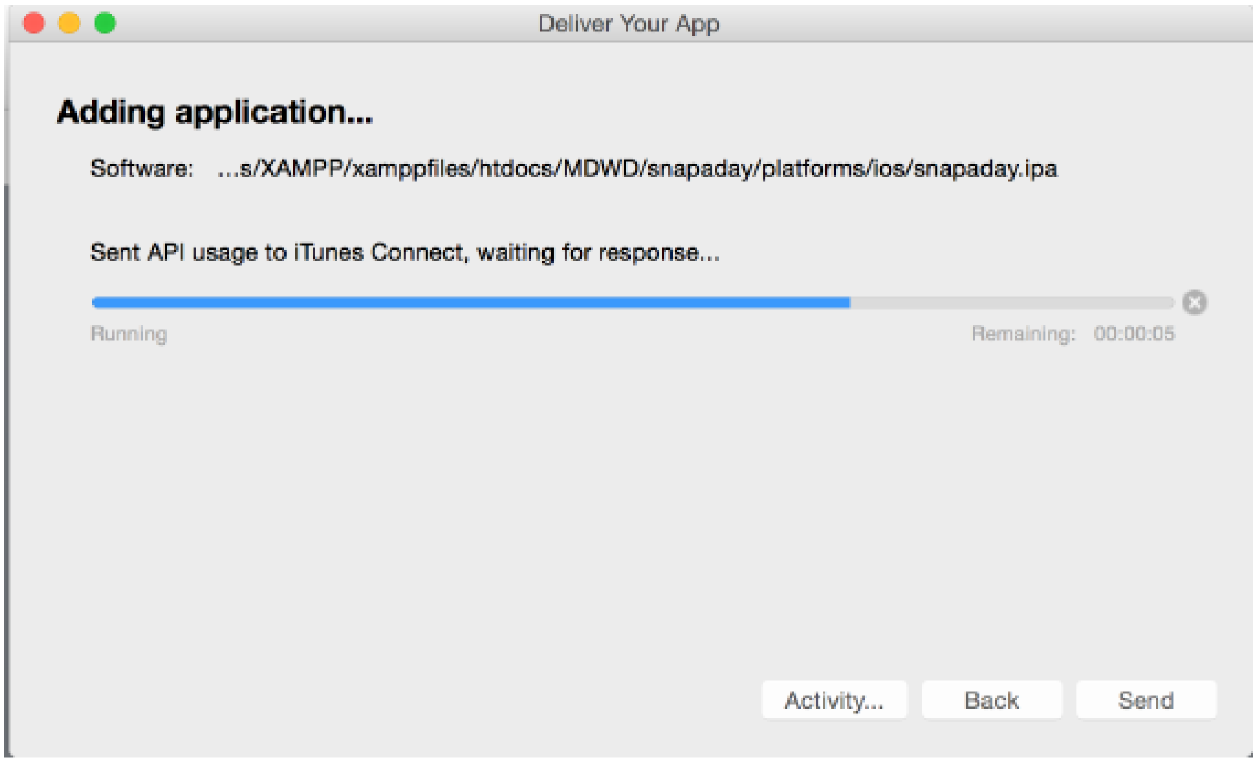
- Macincloud.com allows you to log in remotely to a Mac. This service does cost, but if you just buy some prepaid credits it's pretty cheap and since you will only ever be on there for a few minutes at a time the credit will last ages (this is what I did before I got a Mac).

Once you've figured out how you're going to access Application Loader, open it up and log into your iOS Developer account then choose Deliver Your App:



* Upload the .ipa file that is signed with a distribution certificate and then click Next

Your application will now begin uploading to iTunes Connect:


Deliver Your App

Adding application...

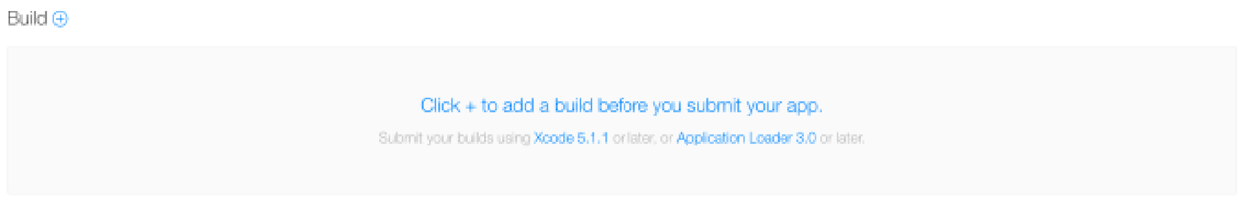
Software: ...s/XAMPP/xamppfiles/htdocs/MDWD/snapaday/platforms/ios/snapaday.ipa

Sent API usage to iTunes Connect, waiting for response...

Running Remaining: 00:00:05

## Submit for Review

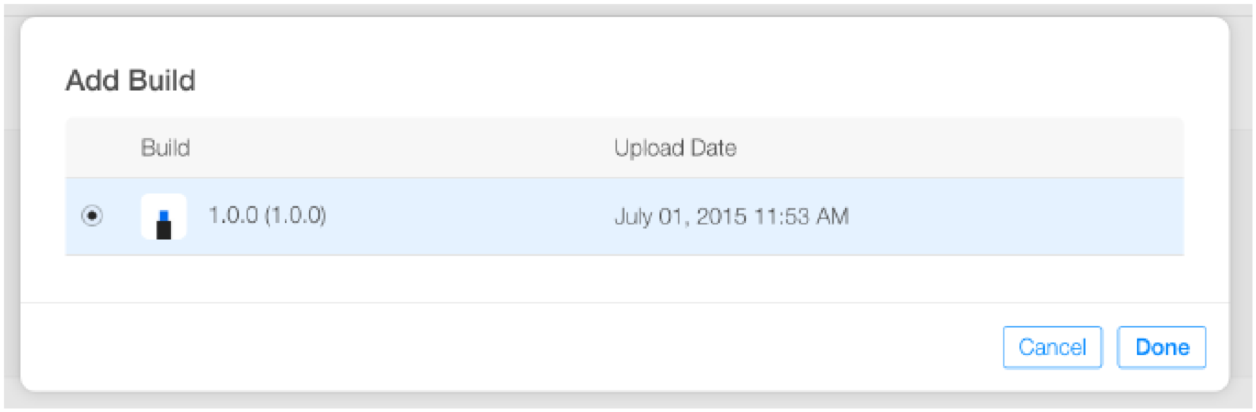
Once you have uploaded your application, either through XCode or Application Loader, you will need to finish up your app store listing in iTunes Connect. Go back to your application in iTunes Connect and go to the **Build** section:



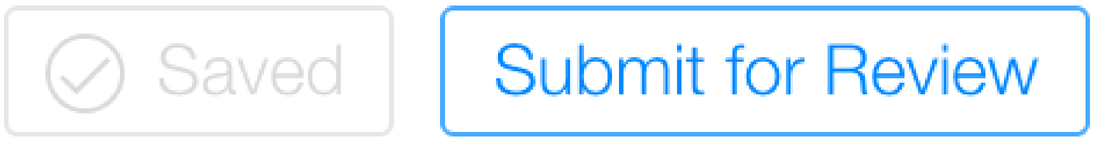
Click + to add a build before you submit your app.

Submit your builds using Xcode 5.1.1 or later, or Application Loader 3.0 or later.

You should now see a **+** icon as shown above. Click this, select the build you just uploaded and hit **Done**:



Double check everything in your listing, then go back to the top of the page, hit **Save** and then **Submit for Review**:



to submit your application to Apple. Now you just have to cross your fingers and wait! The Apple review process usually takes around 5-10 days, which is a frustratingly long time. There's nothing you can do about it though but sit back and wait. Just make sure that you are complying with all of Apples rules and guidelines so that your app does not get rejected (otherwise you will have to fix it and wait another 5-10 days!).

## Submitting to Google Play

If you've been through the nightmare of signing an iOS application and submitting it to the Apple App store then you are in for a treat. Submitting to Google Play is super easy by comparison. Before you get started, you will have to sign up as a Google Play Developer.

Remember, before submitting to Google Play you must sign your **.apk** with a keystore file.

**IMPORTANT:** If you are using the Crosswalk plugin, then you will have two .apk files generated when you build. The process for submitting is still mostly the same, but make sure you read the note at the end of this lesson about how you can upload both .apk files with the same submission.

### Creating a Build for Android

Unlike with iOS, it doesn't matter whether you have a Mac or PC, you will be able to use the same method for both. However, if you are using a PC and are also building an iOS application then that means you will have used PhoneGap Build to do that, since you are already using PhoneGap Build then it makes sense to use it for Android to. So I would recommend using PhoneGap Build to build for Android if you don't have a Mac, unless you are only building for Android.

If you are using PhoneGap Build then you will already have a signed **.apk** file, in which case you can skip to the next section **Submitting your Application to Google Play**. If you do not already have a signed .apk file make sure to follow these steps.

**> Create a file at `platforms/android/release-signing.properties` and add the following:**

```
storeFile=snapaday-release.keystore
keyAlias=snapaday
```

This file tells the build process how we want the application to be signed. Here you will supply the **keystore** file you generated in the signing lesson, as well as the alias of the keystore. The first line should be the path to where the keystore file is stored, for simplicity I move the keystore file to the same location as this file, however you can also specify a different path if you want. The second line is the alias name.

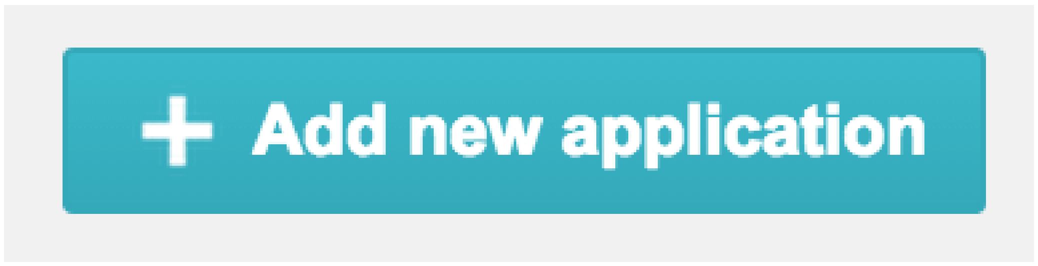
Once you have that file in place, all you need to do is run the following command:

```
ionic build android --release
```

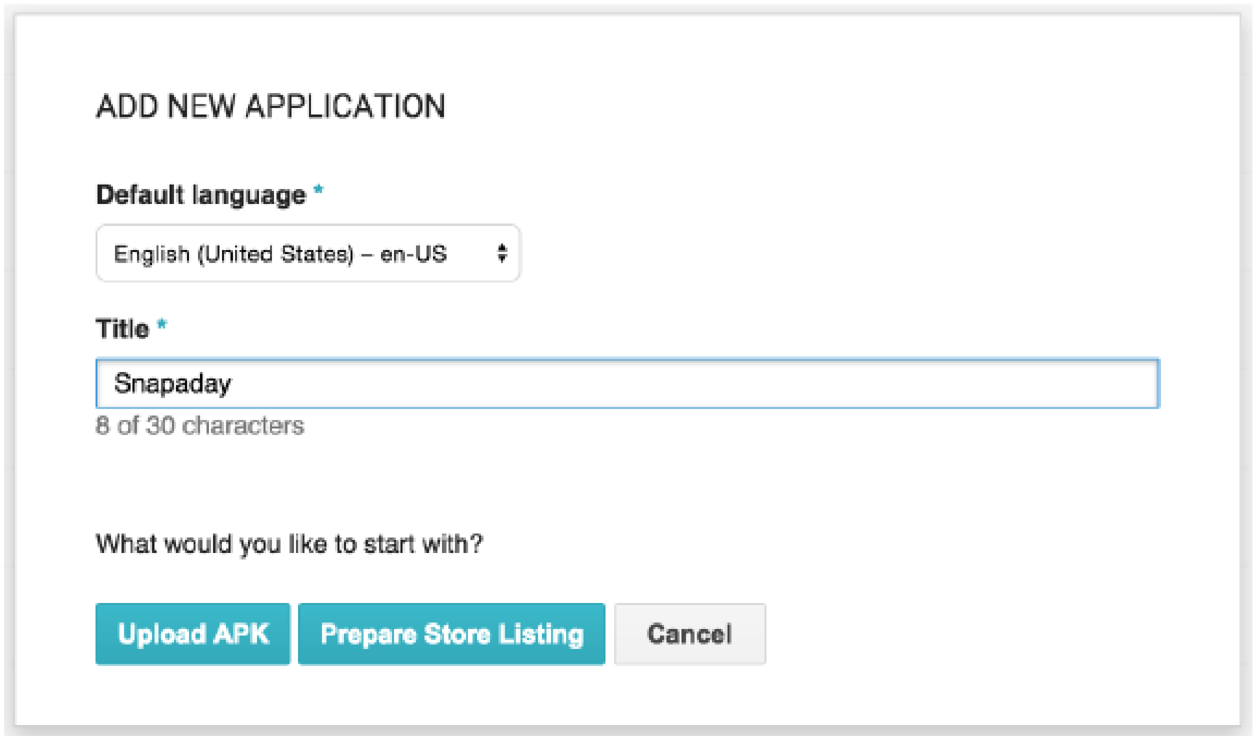
and your **.apk** file (or files if you are using Crosswalk, more on that soon) will be generated for you at: **platforms/android/build/outputs/apk/**

## Submitting Your Application to Google Play

- Log in to the Google Play Developer Console
- Click **+Add New Application:**



* Fill in the information on this prompt and then click **Upload APK**



ADD NEW APPLICATION

Default language *

English (United States) – en-US

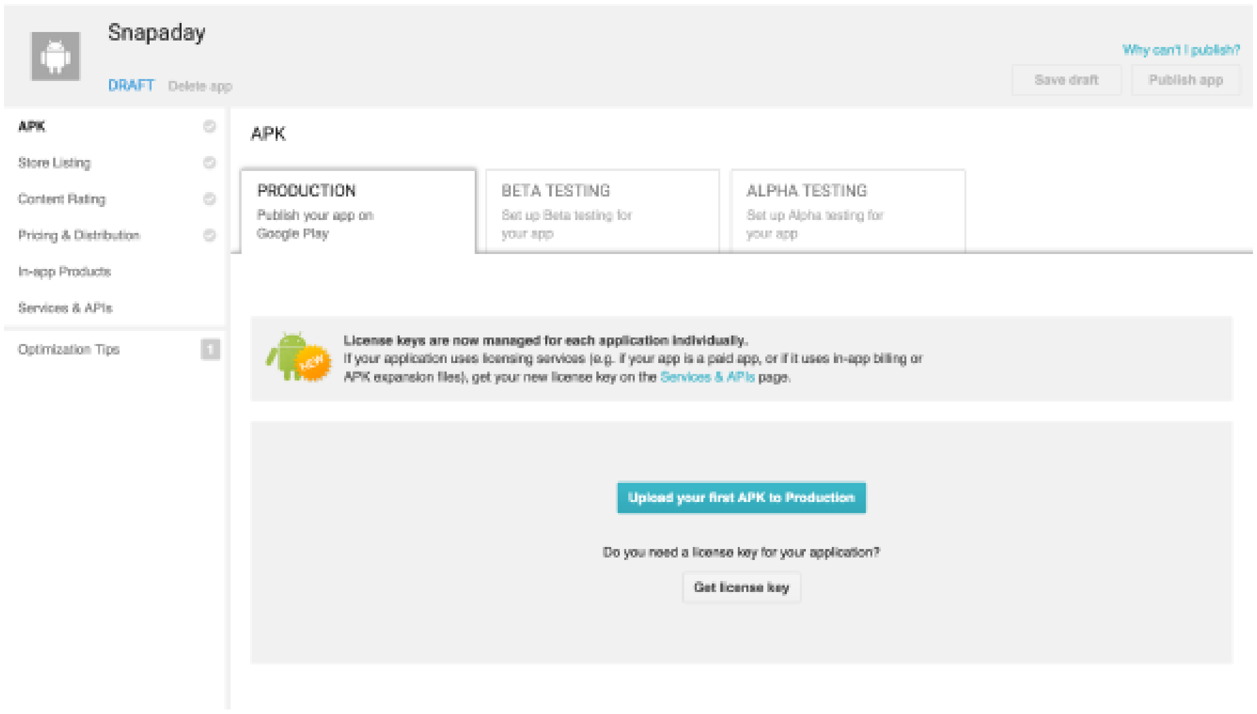
Title *

Snapaday

8 of 30 characters

What would you like to start with?

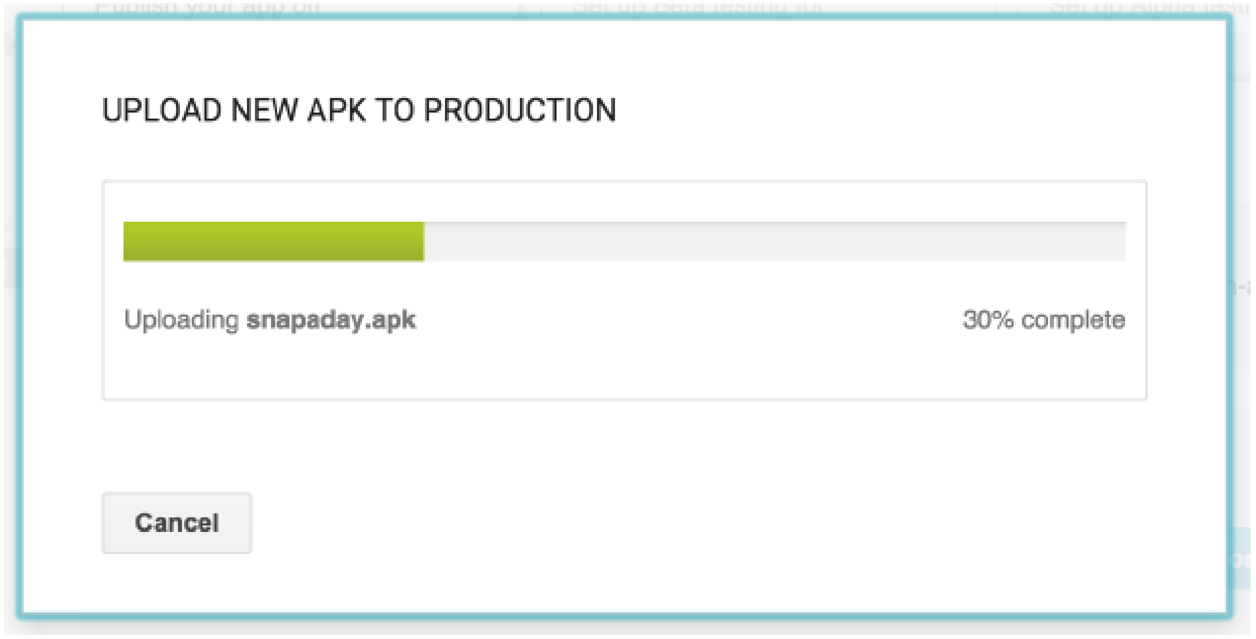
* You should see a page like the following:



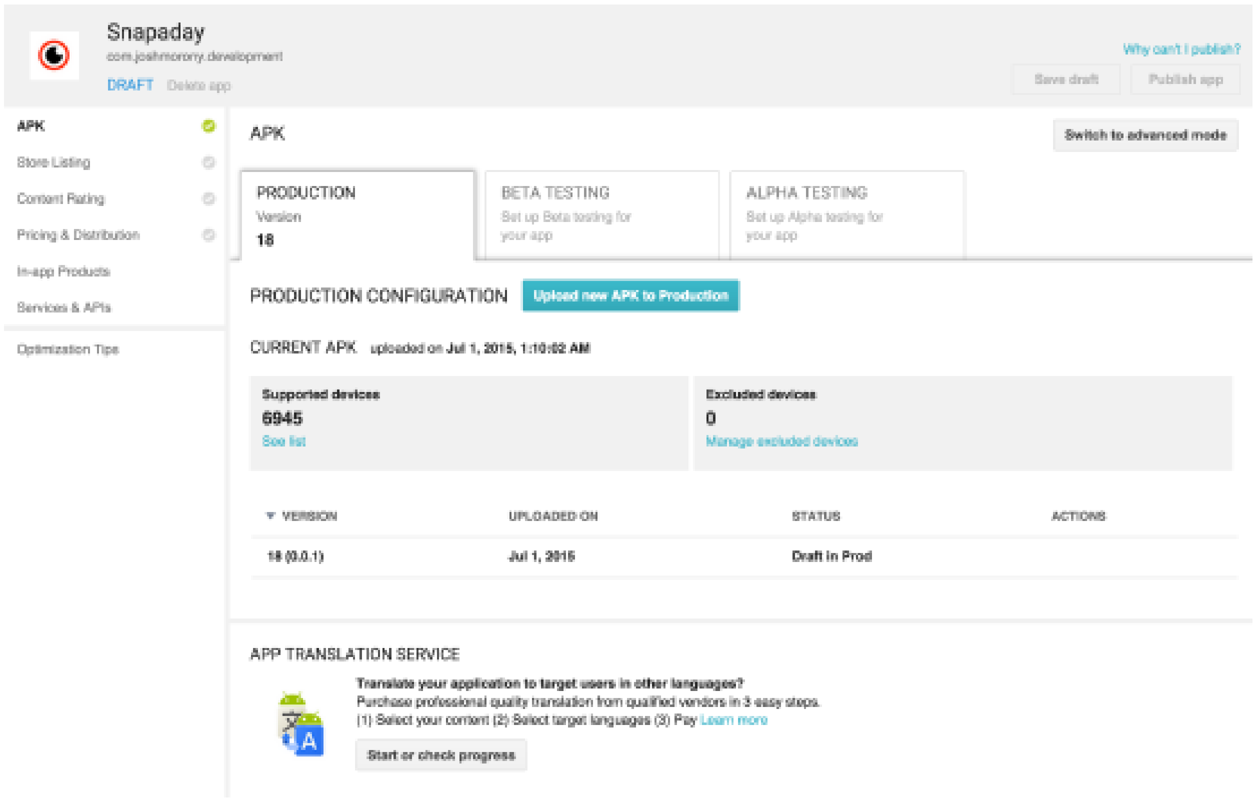
**License keys are now managed for each application individually.**
If your application uses licensing services (e.g. if your app is a paid app, or if it uses in-app billing or APK expansion files), get your new license key on the Services & APIs page.

Do you need a license key for your application?

* Click **Upload your first APK to Production** and upload the signed **.apk** file you created in Lesson 2



* You should now see the page updated like this:



APK

| PRODUCTION | BETA TESTING | ALPHA TESTING |
|---|---|---|
| Version 18 | Set up Beta testing for your app | Set up Alpha testing for your app |

PRODUCTION CONFIGURATION

CURRENT APK uploaded on **Jul 1, 2015, 1:10:02 AM**

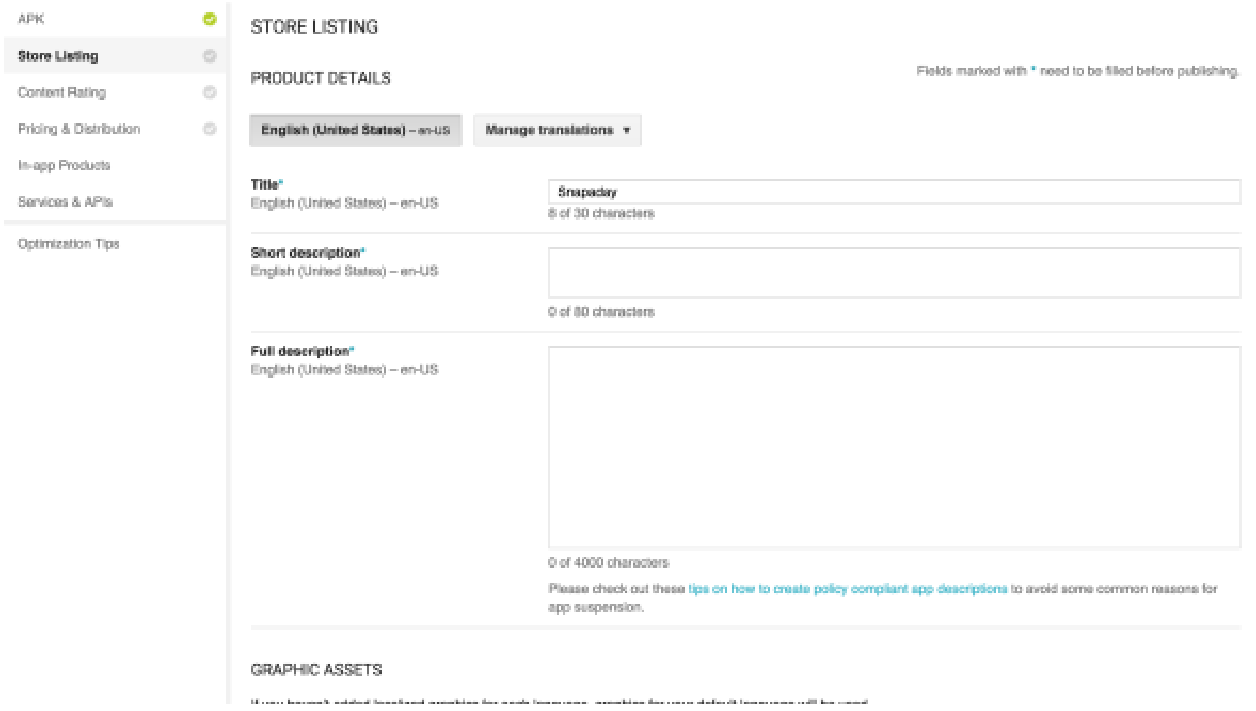
| Supported devices | Excluded devices |
|---|---|
| 6945 | 0 |

| VERSION | UPLOADED ON | STATUS | ACTIONS |
|---|---|---|---|
| 18 (0.0.1) | Jul 1, 2015 | Draft in Prod | |

APP TRANSLATION SERVICE

**Translate your application to target users in other languages?**
Purchase professional quality translation from qualified vendors in 3 easy steps.
(1) Select your content (2) Select target languages (3) Pay Learn more

* Now click on **Store Listing** and fill in all the information including screenshots and promotional graphics for your application then click **Save Draft**



STORE LISTING

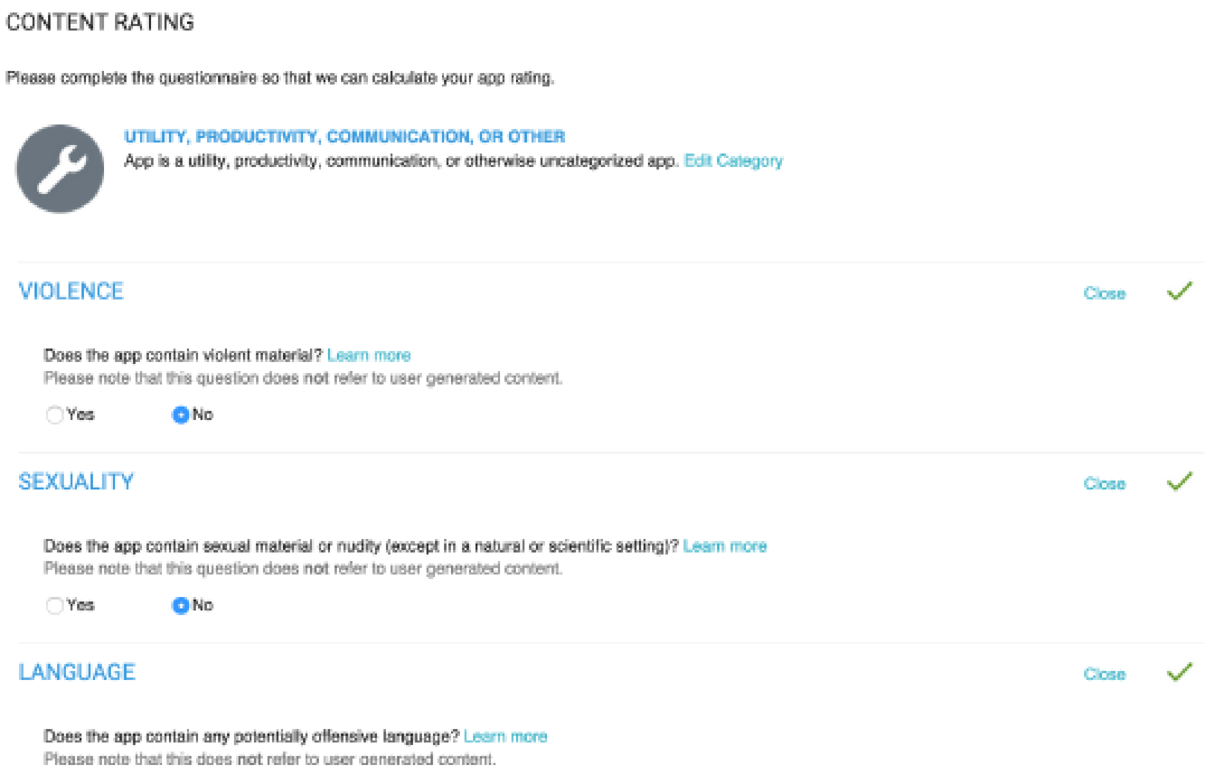
PRODUCT DETAILS

Fields marked with * need to be filled before publishing.

| | |
|---|---|
| **Title*** English (United States) – en-US | Snapaday<br>8 of 30 characters |
| **Short description*** English (United States) – en-US | 0 of 80 characters |
| **Full description*** English (United States) – en-US | 0 of 4000 characters<br>Please check out these tips on how to create policy compliant app descriptions to avoid some common reasons for app suspension. |

GRAPHIC ASSETS

* Next go to **Content Rating** and create a content rating for your application then click **Save Draft**



CONTENT RATING

Please complete the questionnaire so that we can calculate your app rating.

UTILITY, PRODUCTIVITY, COMMUNICATION, OR OTHER

App is a utility, productivity, communication, or otherwise uncategorized app. Edit Category

VIOLENCE — Close ✓

Does the app contain violent material? Learn more
Please note that this question does not refer to user generated content.

Yes [✓ No]

SEXUALITY — Close ✓

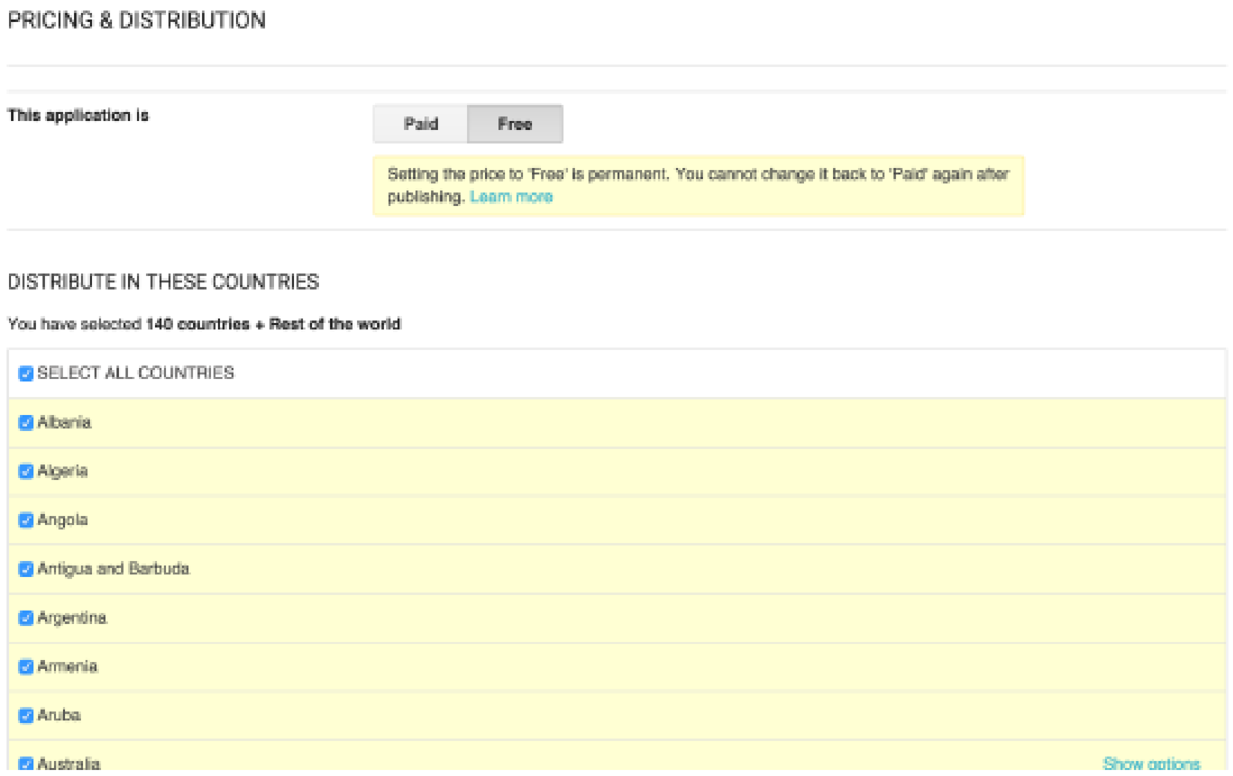
Does the app contain sexual material or nudity (except in a natural or scientific setting)? Learn more
Please note that this question does not refer to user generated content.

Yes [✓ No]

LANGUAGE — Close ✓

Does the app contain any potentially offensive language? Learn more
Please note that this does not refer to user generated content.

* Go to **Pricing & Distribution** and fill out the information there then click **Save Draft**



PRICING & DISTRIBUTION

This application is: Paid [✓ Free]

Setting the price to 'Free' is permanent. You cannot change it back to 'Paid' again after publishing. Learn more

DISTRIBUTE IN THESE COUNTRIES

You have selected **140 countries + Rest of the world**

- ✓ SELECT ALL COUNTRIES
- ✓ Albania
- ✓ Algeria
- ✓ Angola
- ✓ Antigua and Barbuda
- ✓ Argentina
- ✓ Armenia
- ✓ Aruba
- ✓ Australia — Show options

Once you have filled out all of the screens just hit **Publish App** and your application will be submitted!

Unlike the Apple App store, your application should be available on Google Play within hours.

## Uploading Multiple APKs with Crosswalk

The interesting thing about using Crosswalk is that it creates two separate .apk files, which .apk file is required for each user depends on which device they have. If you take a look in your **platforms/android/build/outputs/apk/** folder, you will find the following two release .apk files:

- android-armv7-release.apk
- android-x86-release.apk

This leaves us with a bit of a dilemma - which one do you submit to the app store?

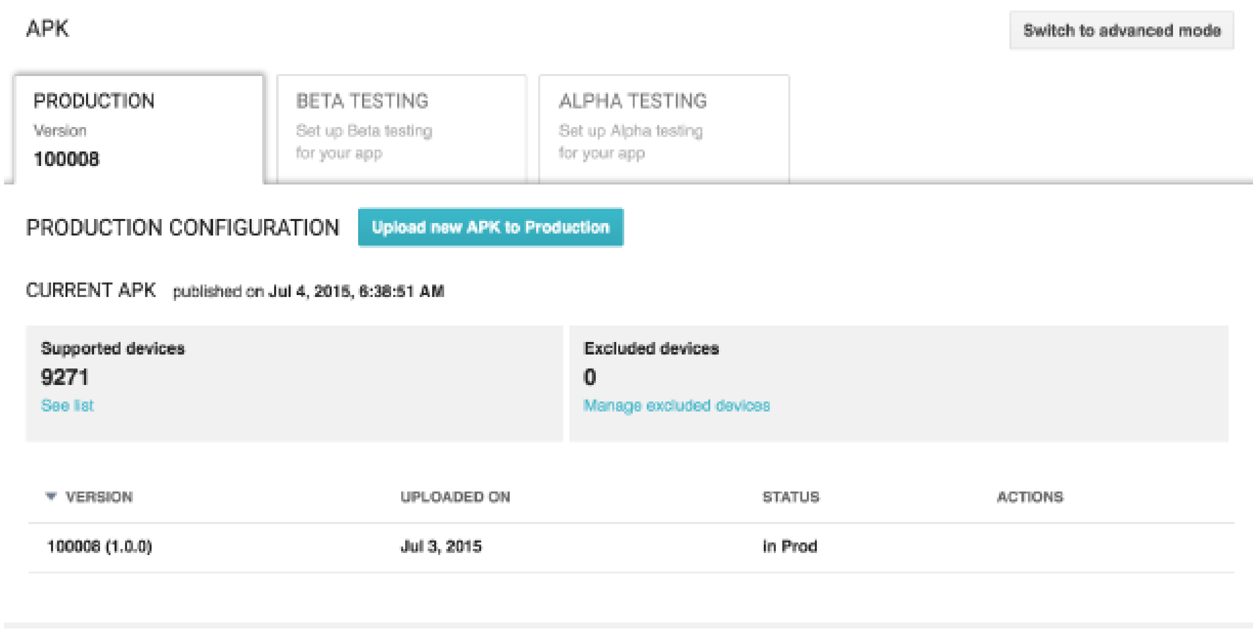
You may read on forums about combining the two .apk files into one, but fortunately the problem is quite easy to solve.

Although it may not seem like it you can actually upload two different .apk files, both the armv7 and x-86 versions, for your app submission (although the ability to do this is hidden slightly).

If you've uploaded apps to the app store before, you may know that whenever you upload a new version of the app it needs to have a higher **version number** than the previously uploaded application (which we can set in the **config.xml** file). This is how you update your application after already having submitted it, you increase the version number in the **config.xml** file, rebuild, and then resubmit.

I'm going to show you how you can upload multiple .apk files for the same submission though - just follow these steps:

- Click on the Upload APK button to upload your first **.apk** file, either **android-armv7-release.apk** or **android-x86-release.apk** it doesn't matter which.
- Once that has finished uploading click **[Switch to Advanced Mode]** in the top right corner of the screen:



| VERSION | UPLOADED ON | STATUS | ACTIONS |
|---|---|---|---|
| 100008 (1.0.0) | Jul 3, 2015 | In Prod | |

Once you have switched to advanced mode, click the upload button again and upload your second .apk file.

You should now have both .apk files uploaded, and be able to see them listed.  You will notice that they have the same version number but a different version code - one version code **must** be higher than the other in order for this to work - fortunately crosswalk handles this for us automatically.

## Updating on the App Stores

You thought you were done? Almost, but there's one more important topic we need to cover.

Once you've got your application live on the app store, it probably won't be too long before you want to make an update - to add some new feature or perhaps to fix a bug that found its way into the production application.

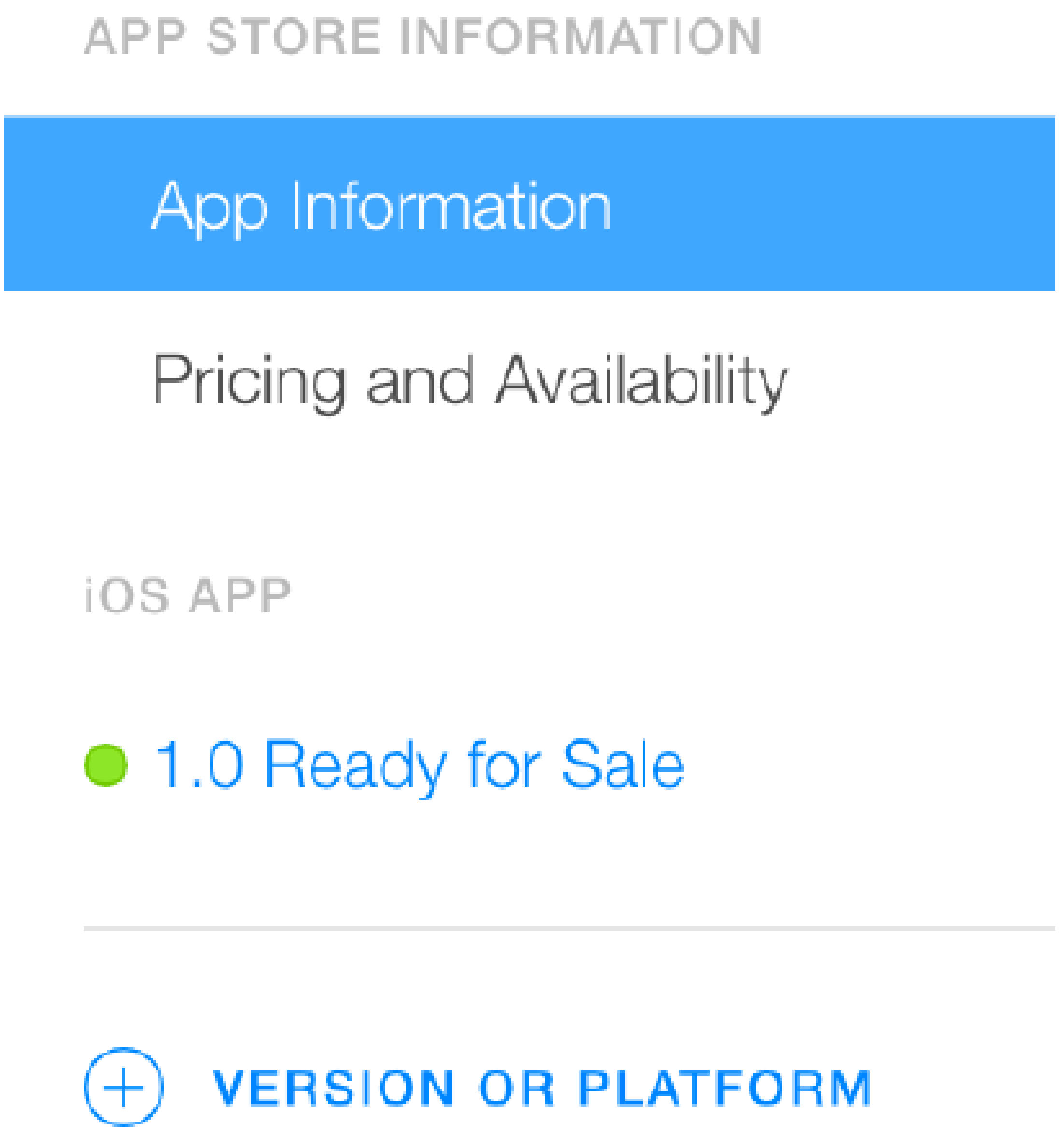
Fortunately, updating your application is really easy. All you have to do is modify your **config.xml** file and bump up the version number:

```
<widget id="com.yourname.yourproject" version="1.0.1"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">
```

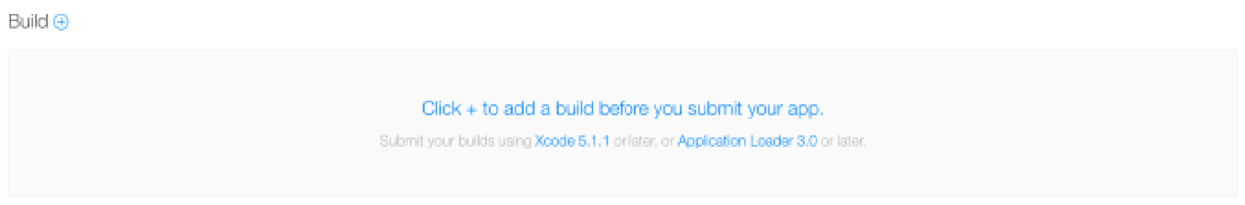
In the example above I've updated the version to **1.0.1** but you could also use **1.1.0** or **2.0.0**, it doesn't matter just as long as the version number is higher than the one you submitted previously.

Then all you have to do is rebuild your application using the exact same steps as you used before, except obviously it will be a bit easier this time because you don't have to generate the certificates and so on again. In fact it's important that you use the same details to sign the application again, otherwise it won't work.

You also won't have to recreate your app store listings of course. In the case of iOS you will just need to log in to manage your existing application in iTunes Connect and upload and then submit a new build. You will find an option to add a new version:



and you should choose the **iOS** option. Enter in the version number that matches the new version in your **config.xml** file and update any fields in the listing that need to be updated. Upload your new build using XCode or Application Loader just like before, and once again add the new build to the app store listing:



Click + to add a build before you submit your app.

Submit your builds using Xcode 5.1.1 or later, or Application Loader 3.0 or later.

Once you have uploaded the new build you can submit it for review again.

For Android you will just need to go to your existing application in the Google Play developer console and update any fields that need updating. You should then go to the **APK** section of the listing and upload your new .apk file (make sure to upload both .apk files if you are using Crosswalk) and then hit ‘Publish now to Production’.

New builds you submit will still need to go through the review process, so for iOS this is going to take about a week again and for Google Play it will be a few hours.

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.

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!