O'REILLY®

Early Release

RAW & UNEDITED

Mobile App Development with Ionic 2

CROSS-PLATFORM APPS WITH IONIC 2, ANGULAR 2 & CORDOVA

Chris Griffith

Mobile App Development with Ionic 2
by Chris Griffith

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-10-25: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491937716* for release details.

978-1-491-93771-6

[FILL IN]

Mobile App Development with Ionic 2

Cross-Platform Apps with Ionic 2, Angular 2 and Cordova

by Chris Griffith

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Who Should Read This Book

This book is for anyone who is looking to get started with the Ionic Framework. It is expected that readers are comfortable with JavaScript, HTML and CSS before starting this book. We will cover some key concepts of TypeScript, ES6, Angular 2, and Apache Cordova, but you may want to have resources available on those topics as well. We will take it step by step, so relax and get ready to learn how to build hybrid mobile apps with Ionic, Angular, and Cordova.

Navigating This Book

This book aims to walk a developer through each part of Ionic, step by step. The book is roughly organized as follows:

- Chapter 1, Hybrid Mobile Apps, is an introduction to the concept of hybrid mobile applications.
- Chapter 2, Setting our development environment, covers what is needed to build Ionic applications.
- Chapter 3, Understanding the Ionic Command Line Interface, digs into the CLI's functions.
- Chapter 4, Just enough Angular and TypeScript, introduces the basic concepts of Angular and TypeScript.
- Chapter 5, Apache Cordova Basics, covers the foundations of Apache Cordova and how it is used as part of the Ionic framework.
- Chapter 6, Understanding Ionic, provides an overview of what makes up an Ionic page.
- Chapter 7, Building our Ionic 2 Do App, we create a Firebase-enabled To Do application
- Chapter 8, Building a Tab-based App, using the tab template to create a National Park explorer application with Google Map integration.
- Chapter 9, Building a Weather Application, build a sidemenu style application the using the Forecast.io weather API and Google's GeoCode API.
- Chapter 10, Debugging and Testing your application, covers some common techniques to resolve issues that can arise during development
- Chapter 11, Deploying your application, walks you through the steps needed to submit your application to the app stores.
- Chapter 12, Exploring the Ionic Services, explores the additional services offered by the Ionic platform.
- Chapter 13, Additional Resources, lists some curated references that can help any Ionic developer.
- Appendix A, Understanding the config.xml fFile, covers the various attributes that configure our application's build process.

- Appendix B, Ionic Components, list each of the available Ionic components and outlines their general use.

The entire code repository is hosted on GitHub, so if you don't want to type in the code examples from this book, or want to ensure that you are looking at the latest and greatest code examples, do visit the repository and grab the contents.

If you have done Ionic 1 development, then you might just want to skim chapters 1 through 3. If you have experience with TypeScript and Angular 2, then feel free to skip Chapter 4. For those who have used Apache Cordova or PhoneGap, then you can bypass chapter 5.

Online Resources

The following resources are a great starting point for any Ionic developer, and should be always available at your fingertips:

- The Official Ionic API documentation
- The Official Angular 2 documentation
- The Official Apache Cordova documentation
- The Ionic Worldwide Slack Channel

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text t hat should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

Tip

This element signifies a tip or suggestion.

Note

This element signifies a general note.

Warning

This element indicates a warning or caution.

Using Code Examples

If you see the ↵ at the end of a code line, this indicates the line actually continues on the next line.

Acknowledgments

To my lovely bride Anita, my twins Ben and Shira, and my good friend Leif Wells.

Table of Contents

Who Should Read This Book. v

1. Forward. 11
2. Hybrid Mobile Apps. 15
3. Setting our development environment. 23
4. Understanding the Ionic Command Line Interface. 37
5. Just enough Angular and TypeScript. 47
6. Apache Cordova Basics. 63
7. Understanding Ionic. 71
8. Building our Ionic 2 Do App. 79
9. 117
10. Building a Weather Application. 153

CHAPTER 1
Forward

In 2013, our small team was then one year into working on drag-and-drop developer tools for the two most popular mobile and desktop web frameworks at the time: jQuery Mobile, and Bootstrap. We saw the rapid rise of reusable components and frameworks for web development and were working hard to make it easier to use them through better and more inclusive tooling.

Around this time, the iPhone 5 came out followed shortly by iOS 7, with dramatically faster web performance and new Web APIs that unlocked previously inaccessible performance and features for browser apps on mobile. We wondered: could a new web framework be built that took advantage of this new performance to provide a very native-like UI kit for web developers to build native-quality apps with standard browser technologies? A "bootstrap for mobile," if you will?

Coincidently, Angular 1 was seeing incredible adoption in the broader web development space and seemed to provide a perfect answer for reusable JavaScript and HTML components for the web. We decided to try our hand at building a new, mobile-first web UI framework and utilize the fast growing Angular 1 framework to make it interactive and distributable.

The first release of Ionic, at the end of 2013, was met with excitement from web developers, and the project quickly accumulated stars on GitHub and installs on NPM. Over the next year and a half, the project saw over one million apps built by fast growing startups, dev shops, and enterprise users alike.

And then, In 2015, JavaScript evolved, seemingly overnight. Suddenly, ES5, the JavaScript we all knew from the Web 2.0 era, was old news. In its place was ES6, the next generation of JavaScript complete with major new features for object-oriented development, sharing and loading modules, easier syntax, and a whole lot more. The Java-

Script world was turned upside down as browser runtimes and developers alike struggled to keep up with the rapid transition to ES6.

Transpilers were built to translate the new JavaScript syntax into the ES5 syntax browsers could understand. Developers experimented to figure out the best way to distribute their JavaScript libraries as reusable modules. New build tools were created, thrown out, and created again, to build and distribute disparate JavaScript modules. New projects, like TypeScript and Flow, took off in an attempt to reduce errors and standardize the syntax of modern JavaScript. Experimental features from ES7 and beyond made their way into transpilers and, much to the chagrin of conservative JavaScript developers, into production code bases, before being deprecated and removed from the standards track. In short: it was chaos.

Framework authors from the pre-ES6 era were suddenly faced with the daunting task of throwing out custom abstractions in exchange for standardized ones now available in ES6 and beyond. Of those frameworks, few had developed such momentum with custom abstractions as Angular 1. For Angular, the question was clear: how do all these domain-specific things like Scope, Controllers, Directives, and the like translate to the world of standardized JavaScript Classes, Web Components, and beyond?

With the rare evolution of JavaScript, the Angular team saw an opportunity to take the lessons learned from building one of the first major JavaScript frameworks and build a framework that would adapt and scale with the future of both web and mobile development. It didn't hurt that the majority of Angular 1 concepts mapped very naturally to ES6 concepts. In fact, in many cases, they felt much more natural in ES6.

When we heard about Angular 2, we knew immediately that it was our own opportunity to take the lessons learned from Ionic 1 and the over one million apps that had been built at the time and build our own framework for the future.

The Ionic team broke ground on Ionic 2 in Spring of 2015. After almost a year and a half of development, mistakes made, novel solutions discovered, and a whole lot of experimentation, we are excited to finally roll out a major, production-ready release of Ionic 2.

At a high level, Ionic 2 is similar to Ionic 1. Components are used by writing custom HTML tags that Ionic turns into powerful mobile components. Actions are bound to callbacks in a class that acts as a controller for a given page in the app. The project is built and tested using the same command line tool. The look and feel and theming is similar and draws on the classic Ionic look from the original release.

What's different is what goes under the hood. In this case, Ionic 2 was rewritten from the ground up using TypeScript and Angular 2. All of the Ionic code is typed, which has dramatically reduced bugs and type issues in our own code and has led to wonderful new features for developers using tools like Atom and Visual Studio Code, such as inline documentation and easy refactoring. Also, the code is more object-

oriented, which just makes more sense for a UI framework. That architecture wasn't as natural in Angular 1.

Angular 2 was rebuilt with the goal of running wonderfully on mobile, both by reducing overhead and streamlining core operations like change detection. Thus, Ionic 2 apps run faster and can handle more complexity than Ionic 1 apps.

The goal of Ionic has always been to be the easiest way to build awesome mobile apps, period. We wouldn't have embarked on a costly and risky rewrite of the framework if we didn't truly believe we could make Ionic easier and more powerful to use at the same time. We believe that TypeScript makes Ionic code easier to write and to reason about. We believe that Angular 2 is easier to use than Angular 1 and requires far less domain-specific language and understanding. We believe that Ionic 2 projects are cleaner and more organized and using components is more straightforward.

In addition to the technologies underneath, Ionic 2 has some major new features. Today, Ionic 2 will adapt the look and feel of your app to match the platform underneath, with much-expanded support for Material Design and easier theming. Our new navigation system makes it possible to build the kinds of flexible and parallel navigations native apps do uniquely well, but which don't have a natural analog in the browser. We've added a plethora of new features, components, and a ton of new Native APIs.

Additionally, the mobile world saw a dramatic shift in 2016. Suddenly, the mobile web is back in vogue as Progressive Web Apps have come onto the scene in a major way. With Google pushing a new world where apps run right in the browser with no install required, and provide a great experience regardless of bandwidth and connectivity, mobile developers are faced with the daunting prospect of adding mobile web as a part of their mobile strategy.

Developers using Ionic 2 can now target the mobile web with practically zero code changes. Ionic apps work both as a native app on iOS and Android, and as a Progressive Web App on the mobile web. Write once, run everywhere!

We've put our heart and soul into Ionic 2, and we're excited to finally be able to recommend Ionic 2 for production mobile apps. We hope you find it just as performant and flexible as we do, and that it makes building mobile apps and mobile websites easier than you ever thought possible. After nearly three million apps built on Ionic, we've learned a thing or two about how to build a quality app framework, and we've taken every lesson learned and put it into Ionic 2.

And, if you like Ionic 2, we hope you take a look at some of the supporting tools we've built to give Ionic developers an edge up, including our rapid testing tool Ionic View, our fast prototyping and rapid app development tool Ionic Creator, and our suite of tightly-integrated backend services with Ionic Cloud. Ionic is becoming a one-stop shop for everything mobile.

From all of us on the Ionic Team, please enjoy Ionic 2 and we hope to see you on the forum!

Max Lynch

Co-Founder/CEO, Ionic

CHAPTER 2
Hybrid Mobile Apps

Mobile application development is becoming one of the most important skills that a developer can possess. Over the past decade, we have seen an explosion of mobile devices; phones, tablets and now wearables, that have given rise to a whole ecosystem of mobile applications. We are now living in an age of mobile apps. But learning how to create them is still a difficult challenge. Typically, a developer will need to learn and master each platform's specific development language; Objective-C or Swift if you are creating iOS-based applications, or Java if you are creating Android based applications. Wouldn't it be nice if there were a solution that allowed for one shared language that we could use across multiple platforms? There is, by leveraging the shared language of the web and some incredible frameworks, developers can now develop their applications in one code base and be able to it deploy to a wide range of mobile platforms. This type of mobile application is known as a hybrid mobile application, where it blends both the native capabilities of the mobile device with the ability develop using web technologies.

What exactly is a hybrid mobile application? Hybrid apps are like any app you will find on your phone. In fact, many of them probably are. The only difference is that those apps are built with web technologies (HTML, CSS, and JavaScript) instead of the device's native development language. A well-written hybrid app shouldn't look or behave any differently than its native equivalent. In fact, users don't care how an application is written, they just want an application that works and looks great.

The Ionic Framework is one of the most popular hybrid mobile application frameworks in use today. The framework has over 26,000 stars on Github and has been forked over 5,700 times. With the release of the next major version of the framework, it is poised to continue its growth as the "go to" solution for hybrid mobile developers.

This book presents the foundations required to build Ionic v2 applications. The goal of this book is to guide you through the process of creating three separate applications. Each of these applications will give you insight into the various components available in the Ionic framework, as well as an understanding of the Ionic ecosystem. Before we get into creating our first application, we need to make sure we have a good understanding of the various foundations the Ionic is built upon, as well as some of the tooling we will be using throughout this book.

What is the Ionic Framework?

So what exactly is the Ionic Framework? Simply put, it is a user interface framework built with HTML, CSS and JavaScript for use with hybrid mobile application development. Beyond just the user interface components, the Ionic Framework has expanded to include a robust command line interface (CLI) and a suite of additional services such as Ionic View, Ionic Creator, and more. We will explore each of these throughout the book.

In reality, Ionic is really a combination of several technologies that work together to make building mobile applications faster and easier. The top layer of this stack is the Ionic Framework itself, providing the user interface layer of the application. Just beneath that is Angular (formally known as AngularJS), an incredibly powerful web application framework. These frameworks then sit on top of Apache Cordova, which allows for the web application to utilize the device's native capabilities and become a native application.

The combination of these technologies enables Ionic to deliver a very robust platform for creating hybrid applications. Each of these technologies will be explored further in this book.

What's new in Ionic 2?

To say that Ionic 2 is a major upgrade is almost an understatement. Not only did the Ionic Framework itself undergo a significant evolution, but one of its underlying technologies, Angular, did as well. Although some things might look the same on the surface, under the hood, there are radical changes. It might be fair to say, that Ionic 2 is almost a new framework. If you are familiar with Ionic 1, much of the component syntax will appear similar, but the code that brings them to life will be new.

Some of the major improvements to the framework include:

Overhauled Navigation: Completely control the navigation experience of your app without being tied to the URL bar. Navigate to any page inside of any view, including modals, side menus, and other view containers, while maintaining full deep-linking capability.

Native Support: There is now more native functionality directly into Ionic, making it easy to take advantage of the full power of the device without hunting down external plugins and code.

Powerful Theming: With the new theming system it's easy to instantly match your brand colors and design.

Material Design: Full Material Design support for Android apps.

Windows Univeral Apps: Support for developing applications that will run on the Windows Universal platform.

But with these improvements to Ionic, comes the added effort of learning the new version of Angular as well as learning TypeScript. We will touch on these requirements in a later chapter.

Comparing Mobile Solutions

When needing to deliver your application to a mobile platform, there are three primary solutions that are available, each with their own strengths and weaknesses. They can be grouped into native mobile applications, mobile web applications, and hybrid mobile applications. We'll look at each solution in a bit more detail to understand the overall mobile application landscape.

Native Mobile Applications

Typically, native code is the solution most developers think of when they need to create a mobile application. To build a native application, the developers need to write in the default language for each targeted mobile platform, which is Objective-C or Swift for iOS devices, Java for Android, and C# or XAML for Windows Univeral.

This type of development comes with several strong advantages over the other options. First, the development tools are tightly integrated with the device platform. Developers are able to work in IDEs that are configured to create mobile applications for that platform, Xcode for iOS development and Android Studio for Android development. Second, since development is done in the native framework, all the native APIs and features are available to the developer without the need of additional bridge solutions. The third advantage is the performance of the application will be as good as possible. Since the application is running natively, there are not intermediate layers of code that can affect performance.

The primary disadvantage of native mobile application development centers around the development language issues. Since quite often you will want to release your application for both iOS and Android (and possibly Windows as well), you will need to have proficiency in all the different languages and APIs. None of the client-side code can be reused, and must be rewritten. In addition, there is a technical burden of maintaining multiple code bases.

Mobile Web Applications

When the iPhone was first announced, there were no third-party applications or even an App Store. In fact, the initial vision was that third party applications were only to be available as mobile web applications and not as native applications. While this is certainly not the case today, creating a mobile web app is still an option. These apps are loaded via the device's mobile web browser. Although the line between a mobile website and mobile app can become blurred, this option is really just about create your application using web technologies and delivering it through the device's browser.

One of the advantages of this solution is that we can have a much wider reach with our application. Beyond iOS and Android, additional mobile platforms become available. Depending on the market that you are targeting, this may be a critical factor. Since you have direct access to your web server, the application approval process that can be tricky or slow at times for native apps is not an issue. Updating your application to add a new feature or resolve a bug, is as simple as uploading new content to the server.

However, the fact these applications run inside the native browser brings along a set of limitations. First, the browser does not have access to the full capabilities of the device. For example, there is no ability for the browser to access the contact list on the device. Another consideration is the discoverability of the application. Users are used to going to their device's app store and finding the app. Going to the browser and inputting a URL is not common behavior.

Hybrid Mobile Applications

A hybrid application is a native mobile application that uses a chromeless web browser (often called a WebView), to run the web application. This solution uses a native application wrapper that interacts between the native device and the WebView. Hybrid apps have a number of advantages. Like mobile web applications, the majority of the code can be deployed to multiple platforms. By developing in a common language, maintaining the code base is easier, and if you are a web developer there is no need to learn a completely new programming language. Unlike mobile web applications, we have full access to the device's features, usually through some form of a plugin system.

However, this solution does have some real disadvantages. Since the application is still just a web app, it is limited by the performance and capabilities of the browser on the device. The performance can vary widely. Older devices often had very poor performing mobile browsers, meaning the app's performance was less than ideal. Although this solution is a native application, communication between the WebView and the device's native features is done via plugins. This introduces another dependency in your project and no guarantee that the API will be available via this method.

Finally, the other native user interface components are not available within the WebView. Your app's entire UI/UX will be completely written by you.

The Ionic Framework takes the hybrid app approach. It aims to provide native looking and feeling UI components (thus addressing that limitation), and by using Cordova and it's library of plugins to handle the access to the device's native capabilities.

Understanding the Ionic Stack

Now that we have a general understanding of the types of mobile application development, let's look a bit deeper into how the Ionic Framework works. Ionic applications are built as part of three layers of technology: Ionic, Angular, and Cordova.

Ionic Framework

The Ionic Framework was first launched in November 2013, and its popularity has quickly grown and continues to grow. Ionic is provided under the MIT license and is available at ionicframework.com.

The primary feature of the Ionic Framework is to provide the user interface components that are not available to web-based application development. For example, a tab bar is a very common UI component found in many mobile applications. But this component does not exist as a native HTML element. The Ionic Framework extends the HTML library to create one. These components are built with a combination of HTML, CSS, and JavaScript, and each behaves and looks like the native controls they are recreating.

Ionic also has a command-line interface (CLI) tool that makes creating, building and deploying Ionic applications much easier. We will be making extensive use of it throughout this book.

The Ionic platform also extends to several add-on services. These include; an online GUI builder you to visually layout the interface of your Ionic applications, push notification system, user analytics, packing and updating solutions. Although many have free developer level access, any production level use will require an additional cost.

The main focus of the Ionic Framework is in the user interface layer and its integration both with Angular and Cordova to provide near native-like experiences.

Angular

The next part of the Ionic stack is Angular (formally known as AngularJS), an open source project primarily supported by Google. Since it's release in 2009, Angular has become one of the more popular web application frameworks in use today. The goal of Angular is to provide an MVW (model-view-whatever) style framework to build complex single page web applications. The Ionic team decided to leverage the power that this framework offers, and built atop of it. For example, Ionic's custom UI com-

ponents are just Angular components. Angular is licensed under the MIT license and is available at https://angular.io/.

With the release of Angular 2, the framework has undergone a massive evolution. This change did cause some discord within the Angular community, but many of the concerns about the changes to the framework have been addressed. We will explore Angular 2 in more detail in a later chapter.

Cordova

The final element of the Ionic stack is Apache Cordova. Originally developed by Nitobi Software in 2009 as an open-source solution to build native applications using web technologies via an embedded Web View. When in 2011, Adobe Systems bought Nitobi, and along with it the PhoneGap name, the project had to be renamed. Although the project was always open-source, the name was not. The open source version was eventually named Cordova (after the street where the Nitobi offices were located). As Brian Leroux, one of the founders of PhoneGap, put it himself: "PhoneGap is powered by Cordova. Think: Webkit to Safari." Adobe continues to be a major contributor to Cordova (along with several other major software companies) and is it licensed under the Apache 2.0 license.

Cordova provides the interface between the WebView and the device's native layer. The library provides a framework to bridge the gap between the two technology stacks (hence the original name of PhoneGap), Much of the functionality is handled through a system of plug-in modules which allows the core library to be smaller. Beyond working on the two primary mobile platforms, Cordova is used on a much wider range of mobile, and some non-mobile platforms, such as Windows Phone, Blackberry, FireOS, and more. For a full list, see https://cordova.apache.org/

Beyond the library, Cordova as has its own command-line tool to assist in the scaffolding, building, and deploying your mobile applications. The Ionic CLI is built atop the Cordova CLI, and we will be making use of it throughout this book.

Prerequisites for Ionic application development

In order to develop Ionic applications, you will need to have some additional technical skills that are not covered in this book. While you do not need to be an expert in these skills, you will need a general knowledge in order understand the concepts of Ionic development.

Understanding HTML, CSS, JavaScript

Since Ionic applications are built using HTML, CSS, and JavaScript, you should have a fundamental understanding of how these technologies combine to build web applications. We will be using HTML to create the foundational structure of our applications. Our CSS will provide the visual styling for our applications. Finally, JavaScript will provide the logic and flow for the applications.

While we will work a modest amount with JavaScript, you will need to be familiar with its syntax and concepts like variable scoping, asynchronous calls, and events.

Understanding Angular 2

Beyond understanding basic HTML, CSS, and JavaScript, you will need some understanding of building web applications. In this book, we will be writing our applications with JavaScript, specifically Angular 2. This means we will be developing in ES6 and writing the code in TypeScript. For many, this is probably something that is new to you. We will cover the basics in chapter 4 to get you up and running.

Access to a Mobile Device

It goes without saying, you are going to need an actual mobile device to install and test your applications on. In fact, you will probably need at least one device for each platform you plan to develop for. While both the iOS and Android SDKs provide emulators/simulators that allow you to see what your app looks like and functions, they are no substitute for testing on a real device.

Summary

Hopefully, you have a better understanding of the difference between the types of mobile application solutions and how the Ionic stack is composed. In addition, you should have a clearer picture of the needed elements for Ionic development.

In the next chapter, we will demonstrate how to set up your computer to develop Ionic applications.

CHAPTER 3

Setting our development environment

One of the initial challenges in developing with the Ionic Framework is the installation and set up of the several tools that Ionic requires. In this chapter, we will walk you through the process of installing all the needed components and the configurations required for developing Ionic applications. The installation process can be broken down into to two main parts: the base Ionic installation, and the platform specific SDK installations. The base installation will cover just the tools that you need to generate your first Ionic application and preview it in your browser. If you want to dive right in and start developing with Ionic, then this is all you will need to do. The second portion of the installation is about setting up your native development environment(s). Even though we are building our apps with web technologies, we will still need to have the native development environments installed on our computers. This will give us access to the emulators and the ability to deploy and test the applications on our devices, and eventually submit them to the app stores.

Throughout this book, we will be using the command line to use the Ionic CLI. On OSX, we will be using the Terminal application. We recommend adding either a shortcut on the desktop or adding it to your Dock. If you are developing on a PC, I personally recommend using GitBash (which can be installed when we install Git) instead of the default command prompt. The command syntax is the same as OSX's, so following along with the code samples should be easier.

Installing Ionic

This section we will get the essential Ionic development environment set up, then generate our first Ionic application and preview it in our browser. You may be wondering why we want to preview our application in a browser. Remember, we are writing our application with web technologies, so it makes sense to target a browser as our first 'platform'. We can leverage browser debugging tools, and iterate through our

development in a much faster fashion. My personal development cycle is to try to stay away from needing to test on a mobile device until I need to.

There are four components we need to install, in the table below you can see the software we need to install on your computer to get started and their urls.

Tool	url
Node.js	nodejs.org
Git	git-scm.com
Ionic	ionicframework.com
Apache Cordova	cordova.apache.org

Installing Node.js

The foundation for Ionic is built atop Node.js (often referred to simply as Node). Node is a platform that enables you to run JavaScript outside of the browser. This has enabled developers to create applications and solutions that are written in JavaScript and can be run almost anywhere. Both the Ionic and Cordova CLIs are written using Node. Because of this requirement, we need this framework installed first.

To install Node, go to http://nodejs.org and download the installer for your development platform. If you already have Node 4.4 installed, you can skip this step. You will want to use the 4.4 version of Node. If you have an additional need to use a later version of Node, you might want to look at Node Version Manager (https://www.npmjs.com/package/nvm) to allow you to easily switch between node versions.

Once Node has been installed, open the Terminal and enter `node -v`. This command tells node to report back the currently installed version.

```
$ node -v
$ v4.6.0
```

If you encounter an issue with the installation, you can review their documentation. With Node successfully installed, we will now install Git.

Installing Git

While you are free to choose any version control solution (Perforce, SourceSafe, or Git), the Ionic CLI leverages Git for the management of templates. In addition, I have found for Windows users, using GitBash is easier to follow along with the examples in this book.

Go to git-scm.com, and click the Download button. Go ahead and open the package file and follow the default installation.

Once the installation is complete, launch the Terminal window as verify it.

In the Terminal, type git --version and press enter.

```
$ git --version
$ git version 2.8.4 (Apple Git-73)
```

With Git now installed on our system, we can install the Apache Cordova CLI.

Installing the Apache Cordova CLI

Although we can install both Cordova and Ionic at the same time, I recommend installing each one individually in case there is an issue during the installation process.

The installation of Cordova CLI uses the Node package manager (npm) to perform the installation. To install it open either your terminal window or GitBash, and enter the following command:

```
$ npm install -g cordova
```

Depending on your internet connection, this can take a bit. For OS X users, you may encounter an issue with permissions during the installation. There are two options; rerun the npm command, but preface it with the sudo command. This will allow the node modules to run as the root user. Alternatively, you can configure your system to solve this permission problem. For details on this visit: http://www.johnpapa.net/how-to-use-npm-global-without-sudo-on-osx/

```
$ cordova -v
$ 6.3.1
```

With these tools in place, we can finally install the Ionic CLI on to our system.

Installing Ionic CLI

Just like the installation of the Cordova CLI, the Ionic CLI is installed via npm as well. In your terminal window, enter the following command:

```
$ npm install -g ionic
```

This install will also take some time to complete. Once the Ionic CLI has completed its installation, we will again check it by issuing the `ionic -v` command in our terminal.

```
$ ionic -v
$ 2.1.0
```

Now we have our base installation in place for Ionic development. However, we are eventually will want to test our applications either in a device emulator or on an actual device. We will take a look at the installation of these tools shortly. But first, let's set up a sample app and see how to preview it in our browser.

Starting a new Ionic Project

The Ionic CLI provides an easy command to enable you to set up a new Ionic project: `ionic start` . This CLI command will generate a basic Ionic application in the active directory. The Ionic Framework can scaffold this project via a collection of starter templates. Starter templates can either come from a named template, a Github repo, a Codepen, or even a local directory. The named templates are *blank*, *sidemenu*, and *tabs*. We will explore these templates later in this book. For now, run the following command to create a new Ionic project:

```
$ ionic start testApp --v2
```

Since we did not define a starter template, the Ionic CLI will default to the tabs template. The CLI will now begin the process of downloading the template and configuring the various components. It may ask you if you wish to create an Ionic.io account. For now, we can ignore this, but we will be exploring the Ionic services later in this book. Once the process is completed, we need to change the active directory to the testApp directory that the CLI generated.

```
$ cd testApp
```

Let's take a quick look at the elements that were installed in this directory.

Ionic Project Folder Structure

The project directory contains quite a number of files and additional directories. Let's take a moment to understand what each item is and their role.

Table 3-1. Ionic Project Elements

src	This directory will contain the actual application code that we will be developing. In earlier version of Ionic v2, this was the app directory.
hooks	This directory contains scripts that are used by Cordova during the build process.
node_modules	Ionic now uses npm as its module management system. The supporting libraries can be found here.
resources	The default icons and splash screens for both iOS and Android are included.
platforms	This directory contains the specific build elements for each installed build platform.
plugins	This directory contains Cordova plugins.
www	This directory contains the index.html that will bootstrap our Ionic application with the transpiled output from the app directory.
.gitignore	A default gitignore file is generated.
config.xml	Used by Cordova to define various app-specific elements.
ionic.config.json	Used by the Ionic CLI to define various settings when executing commands.
package.json	A list of all the npm packages that have been installed for the project.
tsconfig.json	The tsconfig.json file specifies the root files and the compiler options required to compile the project.
tslint.json	TypesScript linter rules.

This is the standard structure of any Ionic 2 project. As we add platforms and plugins to our project, additional sub-directories and files will be created.

Hidden Files

Any file starting with a . (dot) on OS X will not be visible in the Finder.

Changes from Ionic 1 to Ionic 2

If you have used Ionic 1, there are a number of changes that you might want to be aware of. First, Ionic is no longer using Bower for its package management instead, this is now handled through node modules. But the biggest difference is instead of writing your app directly within the www directory, your development is now done in the src directory.

We will explore the various elements in the project folder in a later chapter. For now, let's just test previewing our application in a browser and ensure that we have a working development environment.

Previewing in the browser

One of the advantages of building hybrid applications is that much of the development and testing can be done locally in the browser. In a traditional native application workflow, you would have to compile your application, then either run it in the emulator or go through the process of installing it on a device. The Ionic CLI has a built in command to run the application locally in a browser. With the working directory still the one that was created by the ionic start command, enter the following command: `ionic serve`. This command will start a simple web server, open your browser and load the application for us. It will also listen to changes and will auto-refresh the browser whenever a file is saved.

Setting the port address

ionic serve may prompt you to choose an ip address. In more cases, you should just select the local host choice. If port 8100 is in use, you can select an alternate port by passing in the --p flag followed by the port number you wish to use.

We should now see the starter tab Ionic application running in our browser. The Ionic tab template contains several screens and we can navigate through them and explore some of the various components in the Ionic Framework.

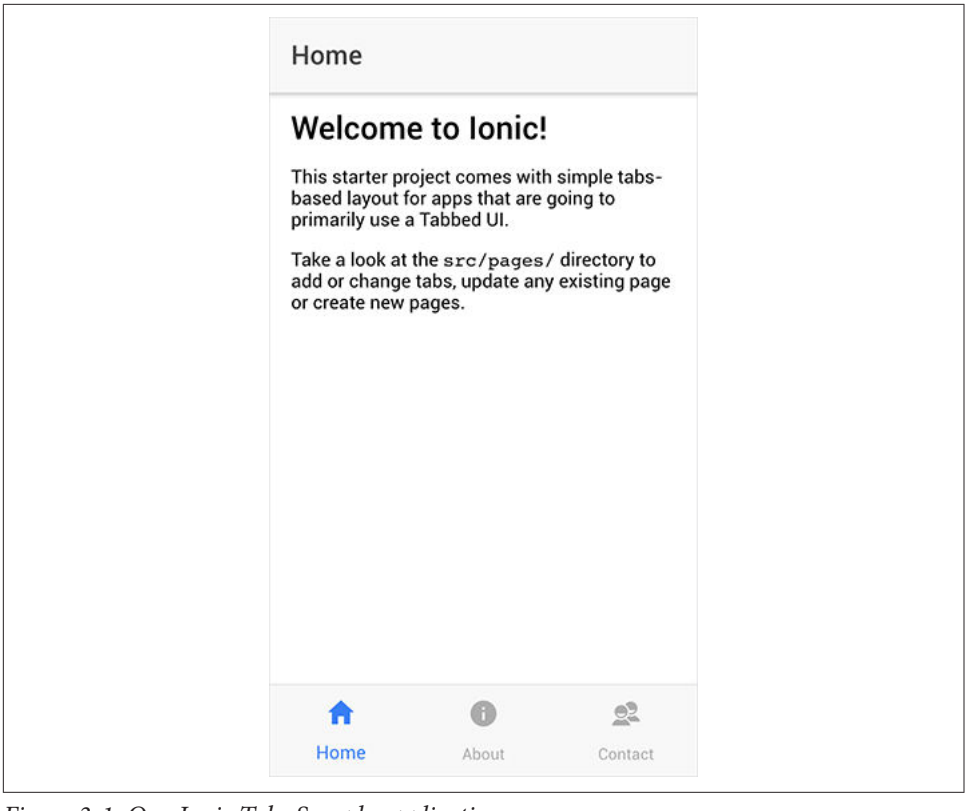


Figure 3-1. Our Ionic Tabs Sample application

Since you are viewing your Ionic app in a browser, you can use all the developer tools that you normally use.

Browser Options

While you are free to use whatever browser you are comfortable with, I recommend sticking with Google Chrome. Although this is not exactly the same browser that your content will run in on your mobile devices, it is similar and you have fewer issues between the desktop testing and the mobile versions.

Platform Tools installations

While we have a working development environment, eventually we will need to continue our development in emulators, and finally on-device. To do this we will now need to install the native application platform tools. This section will be a bit more

complex than the previous installation and specific to each platform we need to install. Thankfully, this is a one-time process; so give yourself to time to complete this section.

Currently, Ionic officially supports iOS, Android, and Windows Universal.

iOS

If you plan to develop for iOS, you will need to use Xcode for both emulation and distribution of your app. Xcode is only available for Macintoshes. While there are some solutions that sidestep the need for a Macintosh (PhoneGap Build and Ionic Build), I recommend having at least a minimum Mac computer available for development.

To install Xcode, simply open the App Store and search for "Xcode". The download is quite large (well over 3 GB), so make sure you have a good internet connection and disk space for installation.

Android

Unlike iOS, development for Android can be done on Windows, Mac and Linux systems. Installation of the Android SDK can be done either via Android Studio or via the stand-alone SDK tools. If you want a complete IDE for Android, then download Android Studio, but we only need the SDK for our development. To download either option, go to http://developer.android.com/sdk/installing/index.html. Personally, we prefer to have the full IDE installed instead of just the SDK.

Installing the Android Studio or SDK will require the installation of the Java Development Kit as well. These additional installation instructions can be viewed at http://developer.android.com/sdk/installing/index.html.

Windows Universal

If you wish to build Windows Universal applications, you will have to do this on a Windows machine. To build applications for Windows Universal, you will download and install Visual Studio 2015 Community Edition from https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx.

During the installation, Select "Tools for Cross Platform Development" as well as the SDK for Windows Universal Apps.

Setting emulators

With the base mobile SDKs installed, we can continue the installation process. For both iOS and Android development, we need to set up the appropriate device emulators. These emulators will allow you to run a virtual mobile device on your computer. We can use these emulators to test our application quickly on various versions of an OS or device type without needing a physical device. They are slower to work with

than directly testing in your browser but can enable the ability to test device specific features, such as working with the Contact database.

Emulators require some additional installation and configuration. Let's look at the steps for each platform.

iOS

Technically, the iOS emulator is a simulator as it does not actually run the native OS, but rather simulates its execution. To install our iOS simulators, launch Xcode, then choose Preferences from the Xcode menu. In the Downloads Tab, you will find a list of available simulators that can be installed. Please note that each simulator is over 1 GB is size, so make sure you have the disk space and internet connection for the download and installation. We typically only have the last two releases installed on my development machine.

Once this installation is complete, we also need to install the Command Line Tools for Xcode. From the Xcode menu, select Open Developer Tool, then the More Developer Tools… option. Then locate the Command Line Tools for Xcode for your version of Xcode and download and install it.

The last piece to work with iOS simulator is to install the ios-sim node module. Open a terminal window and enter the following command:

```
$ npm install -g ios-sim
```

You might need to issue this command with sudo depending on your configuration.

The ios-sim tool is a command-line utility that launches an iOS application on the iOS Simulator.

Now we will be able to run our Ionic apps in the iOS simulator. We will look at this in just a bit.

Android

Before we can configure our Android emulator, we need to install and set up the SDK packages. If you are using the SDK only option, then run

```
<p>$ android <g class="gr_ gr_442 gr-alert gr_spell gr_run_anim ContextualSpelling ins-del multiRe
```

from the command line. This will launch the stand alone SDK manager. This tool will allow you to download the platform files for any version of Android. Like, iOS I recommend only downloading the last two releases packages and tools.

You need to choose the following items:

Tools:

– Android SDK Tools

– Android SDK Platform-tools

– Android SDK Build-tools (choose the most recent version)

Android 6.0 (API 23, when paired with Cordova version 5.4):

– SDK Platform

- ARM EABI v7a System Image

If you are using Android Studio, from the welcome screen, select Configure, then choose SDK Manager. Then install the same components as the stand alone option.

With the SDKs installed, along with the corresponding platform tools, we can now configure the Android emulator. While we can create and configure our virtual android devices within Android Studio, you need to have an active project to do this. Rather, I suggest just using the command line to configure your virtual devices:

```
$ android avd
```

This will open the Android Virtual Device (AVD) Manager. Once it has launched, select the Device Definitions tab, to choose a known device configuration. Select the Nexus 5 definition, then click the **Create AVD…** button. A new window will open with a variety of configurations and additional details that you can set for your virtual device; screen size, which version of Android to run, etc. Once you are satisfied with your device, click OK to finish the process. You can have as many virtual devices as you would like.

Android Emulators

The Android emulators are known to be slow to launch and use. This process has improved greatly in recent releases of the default emulator. However, you might want to look at an alternate solution from Genymotion (https://www.genymotion.com/) for your virtual Android needs.

Setting up your Devices

At some point, you will have to actually test your application on your mobile devices. Each platform has a different set of requirements in order to do this.

iOS

While anyone can test their iOS app in the iOS simulator for free, you must be a paid member of the iOS Developer program in order to test on a device. In the past, provisioning your iOS device for development was a complex process. Thankfully recent changes to Xcode have simplified this process.

First, connect your iOS device to your Mac via a USB cable. Next, we need to create a temporary Xcode project. In Xcode, select New > Project... from the File menu. The New Project assistant will open, then select the Single View Application choice. On the next screen, enter Demo as the Project Name, then click Next. The settings aren't important as we are going to delete this project once we have configured the device. Select a location for the project, then click Create.

Xcode will now display the configuration window for our project. We now need to set the active scheme to our device. This is set via the Scheme control near the top-left of the Xcode window.

With your device unlocked and displaying it's Home Screen, select it from the Scheme dropdown. You should have a warning that there is No Signing Identity Found. Instead of letting Xcode fix this issue, we should manually address it.

In the General settings, in the Identity panel, select your team's name (which is probably just your name) from the drop-down list.

If you do not see your team's name listed, you will need to add your team's account to Xcode. To do this, select the Add Account in the drop-down list. The Accounts preferences window will now open. Enter your Apple ID and password that is associated with your iOS Developer Program account, and click Add.

Once Xcode has finished logging you in and refreshing its list, close the Accounts window. Select your newly added team from the Team drop-down list.

Xcode will now display a new warning about No Matching Provisioning Profiles Found. Click the Fix Issue option and Xcode will resolve this issue.

In order to configure the provisioning profile, Xcode will need some additional information and permissions. You can just answer the question with the defaults.

Let's validate that everything is configured correctly. Click the Run button, located in the top-left of the Xcode window, and making sure that you have your iOS device selected as the target. After a few moments, this test app should launch on your device!

Now, to integrate this into our command line tool chain, we need to install another node tool, ios-deploy. From the command line, enter the following command:

```
$ npm install -g ios-deploy
```

Installation on El Capitan

If you are running OS X 10.11 El Capitan, you may need to add the `--unsafe-perm=true` flag when running `npm install` or else it will fail. For more information on this issue see: https://github.com/phonegap/ios-deploy#os-x-1011-el-capitan

For additional assistance, refer to Apple's documentation at https://developer.apple.com/library/ ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/ .

Android

Setting up an Android device is almost the complete opposite from setting up an iOS device. The first step is to enable Developer Mode on your device. Since each Android device's user interface can vary, these are the general instructions. If you encounter an issue in enabling Developer Mode on your device, review the device's user guide.

1. Open the Settings and scroll to the About Phone item.
2. There should be a Build Number—you must tap on it seven times to enable the developer mode. As you get closer to seven taps, the device should notify you how many taps are left.
3. Once this is complete, you can go back to the Settings list and you'll see a new Developer Options item.

Next, we need to enable USB debugging in order to deploy our apps. In the Developer Options screen, locate the USB debugging option and enable it.

Your Android device is now ready for development. You may be prompt with a dialog to confirm the pairing when you connect the device to the computer for the first time.

Adding Mobile Platforms

Although the Ionic CLI will scaffold much of our application, we might need to add in the target mobile platforms. In order to view our app in either the emulator or on-device, the corresponding platform must be installed. Open your terminal window, and make sure that your working directory is your Ionic project directory. The Ionic CLI command is `ionic platform add [platform name]`.

To add the project files for Android:

```
$ ionic platform add android
```

To add the project files for iOS:

```
$ ionic platform add ios
```

To add the project files for Windows:

```
$ ionic platform add windows
```

By default, the iOS platform is added if you are running on a Mac, so you rarely need to install that platform manually. This command will generate all the needed files for each specific platform.

Previewing on emulator

With a mobile platform added to our Ionic project, we can now verify that we can preview our app in the platform emulator. To run our app in an emulator, use the following command:

```
$ ionic emulate [platform]
```

The Ionic CLI will begin building your app for use on that platform's emulator. You will see a lot of output in the terminal as it goes through the process. Once it has completed it process, the emulator will automatically launch and run your application.

If you need to target a specific emulated device append the command to include the –target switch. For example, if I wanted to emulate an iPad-Air, I would use this command:

```
$ ionic emulate ios --target="iPad-Air"
```

For a list of iOS device types, use this command:

```
$ ios-sim showdevicetypes
```

For a list of Android devices, you will need to refer to the AVD Manager for the device names.

Once you already have the emulator up and running, you can run the `emulate` command again without closing the emulator. This is faster than exiting the emulator and relaunching it every time you change files because the emulator doesn't have to reboot the OS each time.

The Ionic CLI has another very powerful feature that allows you to reload the app instantly using the live reload flag, **--livereload**. The feature was a huge timesaver when you are working with emulators during our Ionic 1 development workflows. However, recent device security changes have currently disabled this feature. It is not clear if a solution will be found.

You also can output the console logs to the Terminal so you can read them easier. See the blog post about the feature at http://blog.ionic.io/live-reload-all-things-ionic-cli/.

```
$ ionic emulate ios -l -c

$ ionic emulate android -l -c
```

Previewing on device

Although the emulators do a fine job, eventually you will need to install your application on a physical device. The Ionic CLI makes it extremely easy to do so with the run command. In your terminal, enter ionic run [platform].

The Ionic CLI will then begin the process of compiling the app for installing the app on a device, then install it on the connected device. Be aware that each installation will overwrite the existing installation of the app.

For iOS deployment, you will need the ios-deploy node module installed as well; otherwise, you will have to manually perform the installation via Xcode.

```
$ ionic run ios -l -c
```

If you are developing for Android on a Windows machine, you might need to download and install the appropriate USB driver for your device. Check https://developer.android.com/tools/extras/oem-usb.html to see if one is needed for your device. No additional drivers should be needed for OS X in order to deploy to an Android device.

```
$ ionic run android -l -c
```

If a device is not found, the Ionic CLI will then deploy your app to an emulator/simulator for that platform.

Summary

This chapter covered all the steps needed to set up your Ionic development environment. In this chapter, we built a first test app and previewed it locally in our browser, in an emulator, and finally on our device.

CHAPTER 4

Understanding the Ionic Command Line Interface

One of the key tools that we will be using while developing our Ionic applications is the Ionic command line utility or CLI. We touched briefly on this tool during our initial setup, but this chapter will explore the various options this utility gives us.

First, if you have not installed the Ionic CLI, you can use npm to do so. If you are on a Macintosh, launch the Terminal application. For Windows users, launch Git Bash (or another command prompt). Then enter the following command:

```
$ npm install -g ionic
```

This will install the latest version of the Ionic CLI. The Ionic CLI is fully backward compatible with version 1 projects if you have already done some Ionic development.

Note:

Mac OS X users might need to prepend the npm command with sudo for the installations to work.

Once the installation of the Ionic CLI is complete, we can test its installation by building our first test application.

```
$ ionic start myApp [template name] --v2
```

This command will create a new Ionic application in a new directory named myApp using the template that we pass to the command. Let's explore the various template options that we can use.

Ionic currently has three named starter templates: *blank*, *sidemenu*, and *tabs*. If no template is passed as a parameter, then the tabs template will be used.

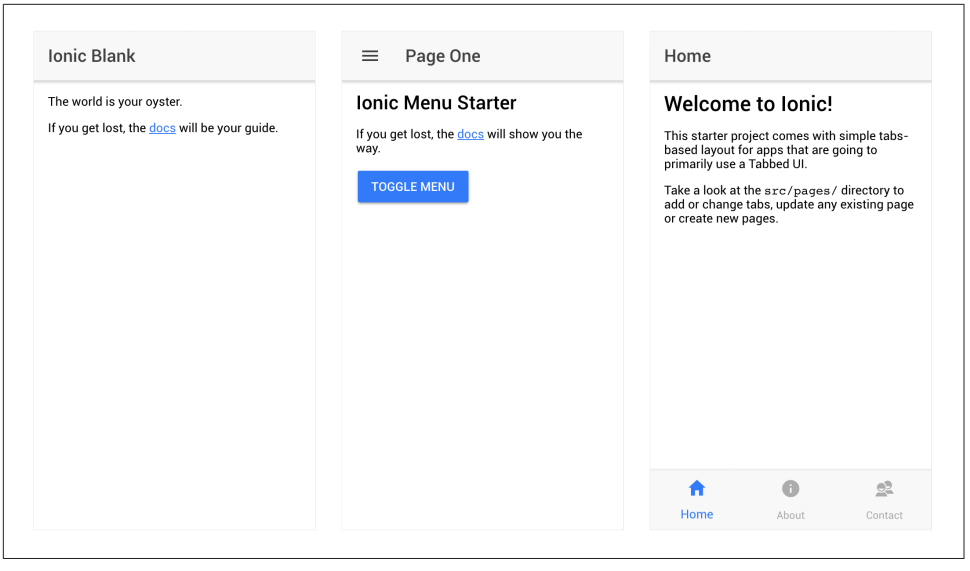


Figure 4-1. The Ionic templates: blank, sidemenu, tabs.

Besides using the three named templates, you can also pass an url to your own template that could be hosted at GitHub, CodePen or some other web address. In fact, these named templates are actually just aliases to repos on GitHub. Here is an example of creating an Ionic app using a template from a CodePen.

```
$ ionic start myApp http://codepen.io/ionic/pen/odqCz
```

If you do not want to host a template externally, you can also pass in either a relative or absolute directory as a source location of your template files.

Since the Ionic CLI supports both version 1 and version 2, we need the ability to tell it which version to create. This is done using the **--v2** flag. By passing this flag to the command, the Ionic CLI will scaffold our Ionic application to use version 2 of the framework.

```
$ ionic start myIonic2App --v2
```

The --ts flag

If you used earlier versions of Ionic 2, you might be familiar with the --ts flag, which told the CLI to use TypeScript as the development language. Starting with beta 8, the decision was made to only support TypeScript. Thus, the need to include this flag was removed.

There are some additional command line flags that you can pass to the command as well. By default, the Ionic start command will take the name of the directory that is

created and set that as the app's name in the config.xml file. You are free to change this within the config.xml at any time, but instead you can use either the –appname or –a flag followed by the actual appname. Since your appname will probably include spaces, your will need to use quotes around the appname for this to work.

```
$ ionic start myApp -a "My Awesome Ionic App"
```

Another flag you will wish to change is the app id. App IDs are the unique name that each platform uses as to identify each app. The recommended format for an app id is a reverse-domain style naming. By default, the Ionic start command will auto generate an app id in the format com.ionicframework.[your appname] + a random number. I doubt that this would be the app id you would want your app to be known by the internals of the various app stores. To change this you can use either the --id or –i flags followed by the package name. This will change the id attribute in the widget node of the config.xml file.

```
$ ionic start myApp -i com.mycompany.appname
```

The last flag that you can include is the `--no-cordova` or `–w` flag. This flag will tell the ionic start method not to include any of the Cordova elements when creating the project. You might be wondering why would you even want to do something like that? Aren't we building mobile applications and need to use Cordova as part of the process? Yes, however, you might want to use Ionic as the framework for a mobile web only application, so you could not use Cordova. Another option might be using Ionic as the UI framework for an Electron-based desktop app (http://electron.atom.io/). In either case, it is an option that is available if you need it.

Define your Build Platforms

Once the base Ionic application has been scaffolded, we next need to add the target platform we will want to build for. The command to do this is:

```
$ ionic platform add [platform name]
```

Common Mistake

If you try to run this command without changing your active directory into the project directory that was just created you will get an error. Always remember to issue cd [appname] before running any of the other Ionic CLI commands.

Mac OSX users will have the iOS platform automatically added to the project. This platform is not available if you are running the CLI on Windows. But if you ever need to manually add the iOS platform, the command is

```
$ ionic platform add ios
```

To build for the Android platform, you will need to add it to the project.

```
$ ionic platform add android
```

To build for the Windows Universal platform, you will need to add it to the project.

```
$ ionic platform add windows
```

Remember for iOS, Android, and Windows Universal, their respective SDKs must be installed on the local machine in order to actually build for that platform. This Ionic command only configures the local project files for use by the SDKs.

If for some reason you need to remove a platform from your project, you can use the following command:

```
$ ionic platform remove [platform name]
```

Occasionally, something might go wrong during an installation of a plugin or during an update. One common solution is to remove the platform, the reinstall it into the project.

Managing Cordova Plugins

The installation of Cordova plugins is often one of the first things you will do after including your build platforms. Although we will touch upon using Cordova plugins and Ionic Native in a later chapter, the basic command is

```
$ ionic plugin add [plugin id]
```

Usually, the plugin id is the npm domain name, for example, cordova-plugin-geolocation. However, it could reference a local directory or a Github repo.

To remove an installed plugin, the command is simply:

```
$ ionic plugin rm [plugin id]
```

If you ever need to see a listing of all the plugins you have installed in your project, use this command:

```
$ ionic plugin ls
```

Ionic Generator

Although the Ionic CLI will scaffold your application via the **ionic start** command, you can then extend your app with new pages, components, providers, pipes and more via the CLI. The **generate** command allows you to quickly create a boilerplate version of the element you need. The command is:

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

For example, if we want to create a new page for an application, the command is simply:

```
$ ionic g page myPage
```

The Ionic CLI will then create a new directory in our app directory, and generate the HTML, SCSS, and TS files for us.

A quick note on naming conventions

Ionic 2 uses kebob-casing for file names (my-about-page.html) and css classes (.my-about-page), and uses PascalCasing for JavaScript classes in TypeScript (MyAboutPage).

Previewing your application

Often early development can be previewed and tested locally in a browser. Although the Ionic CLI does make it fairly easy to launch your application in an emulator or on an actual mobile device, the ability to preview and debug in what is probably a very familiar environment, your browser is a real advantage.

The ionic serve command will start a local development server, then automatically load your application into the browser. In addition, the command will start a LiveReload watcher, so as you make changes to the application, it will be automatically reloaded in the browser without you needing to manually reload it. The command is:

```
$ ionic serve
```

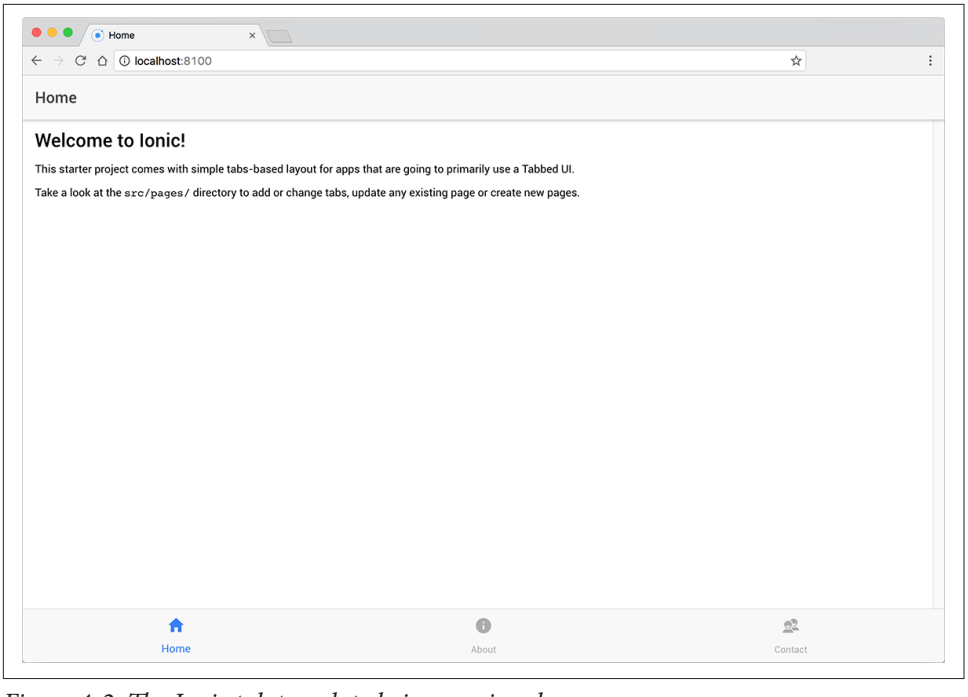
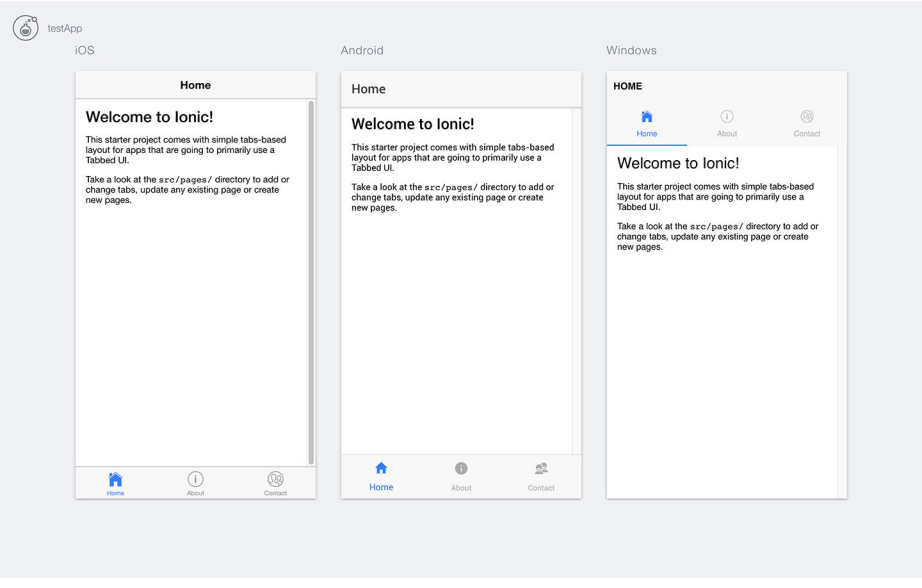


Figure 4-2. The Ionic tab template being run in a browser.

The Ionic Lab Command

With the --lab flag passed to ionic serve, your application will be displayed in an iOS frame, an Android frame and a Windows frame in the same browser window. This feature will let you quickly see the various platform specific differences that may exist. To use this simply type:

```
$ ionic serve --lab
```



Ionic serve running in --lab mode.

In this mode, each instance of the Ionic app will run as if on that mobile platform. So any platform specific CSS or JavaScript will be executed. This feature can be a real time saver during the early part of the development cycle, but it is in no way a substitute for testing on actual devices.

Specifying an IP Address to use

If you need to specify what address the LiveReload server will run on, you can pass that value via the --address flag, followed by the IP address. For example

```
$ ionic serve --address 112.365.365.321
```

Emulating Your Ionic App

The ionic emulate command will build your Ionic application and then load and run your app on the specified emulator or simulator.

```
$ ionic emulate android
$ ionic emulate ios
$ ionic emulate windows
```

Emulators are useful for testing portions of the application that require actual device features. Launching and previewing your application in an emulator takes some time to initialize and then load your app.

Emulating iOS devices

In order for the Ionic CLI to communicate with the iOS simulator, an additional node package will need to be installed. If you did not do this step in the previous, please do it now.

```
$ npm install -g ios-sim
```

Once this package is installed, the Ionic CLI will be able to compile the app and run it within the iOS simulator. If you need to target a specific iOS model, you can set the --target flag to the specific device.

For a list of your installed devices, use

```
$ ios-sim showdevicetypes
```

iOS Simulator Device Types

iPhone-5, 10.0	iPhone-6s-Plus, 10.0	iPad-Air-2, 10.0
iPhone-5s, 10.0	iPhone-7, 10.0	Apple-TV-1080p, tvOS 10.0
iPhone-6, 10.0	iPhone-7-Plus, 10.0	Apple-TV-1080p, tvOS 9.1
iPhone-6-Plus, 10.0	iPad-Retina, 10.0	Apple-Watch-38mm, watchOS 2.1
iPhone-6s, 10.0	iPad-Air, 10.0	Apple-Watch-42mm, watchOS 2.1

Supported iOS Devices

Although both the Apple-TV and Apple-Watch are listed by ios-sim, these platforms are not supported by Apache Cordova nor Ionic.

Emulating Android devices

To emulate your Ionic application in the Android emulator, you first must have manually created an Android virtual device (AVD) to be used by the emulator. If you did not do this in the previous chapter use the command

```
$ android avd
```

This will launch the AVD manager, a tool that you can use to create and manage various AVDs. Once you have created an AVD, the Ionic CLI will be able to launch the Android emulator and run your application. This process can take quite some time as the emulator boots up.

To target a specific Android device, you can use **--target=NAME** to run the app in the specific device you created; otherwise, the default emulator is used.

Performance Tips

If you are using the Android emulator, one tip for improved performance is not to close the emulator, but keeping it running instead, and just reloading the app.

Although the performance of the emulator has improved, many developers have opted to a solution from Genymotion as an alternate to the standard Android emulator. To learn more about this solution, visit https://www.genymotion.com for more information.

Running Ionic App on a device

The Ionic CLI can also compile your Ionic application so it can run on a properly configured device. The command is

```
$ ionic run [platform name]
```

If no device is found to be connected to the computer, the Ionic CLI will then attempt to deploy to the emulator for that platform.

If you are deploying to an iOS device, there is an additional node module that you will need to install, ios-deploy. To do this, run this command:

```
$ npm install -g ios-deploy
```

In addition, to having to install the node module, your iOS device must be configured for development.

If you are deploying to an Android device, you only need to set that device into development mode.

Logging

Both the emulate and run commands support the ability to remap the console logs, as well as the server logs to the Ionic CLI. To enable console logging, pass in either the --**consolelogs** flag, or the short version --**c**. If you want to capture the server logs, pass in the --**serverlogs** flag, or the short version --**s**.

CLI Information

If you ever need to see the entire state of the Ionic CLI and its supporting tooling, use this command:

```
$ ionic info
```

Here is what my system looks like:

```
Your system information:

Cordova CLI: 6.3.1
Ionic CLI Version: 2.1.0
Ionic App Lib Version: 2.0.0-beta.20
ios-deploy version: 1.9.0
ios-sim version: 5.0.8
OS: Mac OS X El Capitan
Node Version: v4.6.0
Xcode version: Xcode 8.0 Build version 8A218a
```

Often when debugging an issue this information can be quite useful.

Summary

In this chapter we have touched on the principle Ionic CLI commands that you will typically use during your development process. There are addition commands and settings available. To see the full list, use *ionic --help* for a complete list.

To recap the key commands that we introduced in the chapter

- Learn to scaffold your initial Ionic application with `ionic start`
- Manage mobile platforms to the project with `ionic platform`
- Preview the application in the browser, emulators, and on-device with `ionic run/emulate`

CHAPTER 5

Just enough Angular and TypeScript

With our basic system configured for Ionic development, we can explore another set of foundational technology that Ionic is built atop, Angular. Just as Ionic leverages Apache Cordova to act as the bridge to the native mobile platform, it uses Angular as the underpinnings of its interface layer. Since the beginning, the Ionic framework has been built on top of the Angular framework.

Why Angular 2?

Angular 2 is the next version of Google's incredibly popular MV* framework. This new version of Angular was announced at the ngEurope conference in October 2014. The Angular team revealed that this version of Angular would be a significant revision. In many ways this new version of Angular was a completely new framework, sharing only its name and some notional references to the original version. This announcement certainly generated a lot of concern and uncertainty about the future of the framework. This was equally true with the Ionic community and within the Ionic team itself. What were the changes that would be required? How much relearning would existing Ionic developers need to undertake to continue to work with Angular?

But as the shock wore off, it became clearer that this evolution of the Angular framework was for the better. The framework was becoming faster, cleaner, more powerful and also easier to understand. The Angular 2 team also took another bold gamble and looked at the web to come, and not the web that was. So they also decided to embrace many of the latest and emerging web standards and develop it in next generation of JavaScript.

At ngConf 2016, the Angular team announced that with release candidate 1, the framework will be simply known as Angular.

So let's take a look at some of these changes to Angular in more detail.

Components

One of the biggest changes from Angular 1 to Angular 2, was the fact that we no longer rely on scope, controllers, and to some degree directives. The Angular team adopted a component-based approach to building elements and their associated logic. For those who have developed applications with more traditional frameworks are very familiar with this type of model. The fact is that we are developing on a platform originally designed to read physics papers and not a platform to build applications upon.

Here is what a sample Angular component looks like:

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

@Component({
  selector: 'my-first-component',
  template: `<div>Hello, my name is &#123;{name}&#125;.
  <button (click)="sayMyName()">Log my name</button></div>`
})

export class MyComponent {
  constructor() {
    this.name = 'Inigo Montoya'
  }
  sayMyName() {
    console.log('Hello. My name is ',this.name,'. ↵
                You killed my father. Prepare to die.')
  }
}
```

This is the exact same model that Ionic uses to generate its component library. In fact, there is nothing that prevents you from extending your Ionic application to use your own custom components.

Let's look at this code snippet in greater detail.

First, the code imports the Component module from the Angular library. In Angular this is how dependency injection is handled. With Release Candidate 1 (RC1), the Angular team broke the library into smaller modules, as well as dropped the 2 suffix in the library.

Next, we use the @Component decorator to provide some metadata about our code to the compiler. We define the custom HTML selector to use. So when we use `<my-first-component></my-first-component>`, the associated template will be inserted into the DOM. Templates can come into two fashions; inline as shown here or as an external reference. If you need to span your template across multiple lines for readability, make sure you use the backtick (`) instead of a single quote (') to define the template string. We will look at templates in more detail later in this chapter.

After the decorator, we export the class definition itself, MyComponent. Within this constructor of this class, the code sets the name variable to 'Inigo Montoya'. Unlike with Angular 1, and JavaScript in general, Angular 2 has a much tighter control over the scope of variables.

Finally, this sample class has a public method of sayMyName that will write a string to the console. As you work more with Ionic and Angular 2, this new method of creating components and pages will become more familiar and natural to you.

Inputs

Since Angular 2 is built using a component model, it needs a mechanism to pass information into the component itself. This is handled via Angular's Input module. Let's look a simple component, <current-user> that will need to know about a user argument in order for the component to perform its code. The actual markup would look like this

```
<current-user [user]="currentUser"></current-user>
```

while the component itself would look like this

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

@Component({
    selector: 'current-user',
    template: '<div>&#123;{user.name}&#125;</div>'
})

export class UserProfile {
    @Input() user;
    constructor() {}
}
```

Within the class definition, there is now a @Input binding to the user variable. With this binding in place, Angular will pass in the currentUser variable into the component, thus enabling the template to render out the user.name value.

This is how Ionic's component also function. We will pass in data and configuration parameters using the same system as this example.

Templates

Templates are HTML fragments, that Angular combines with specific elements and attributes to generate the dynamic content. For the most part, the templating system in Angular 2 did not change that much.

{}: Rendering

```
<div>
 Hello, my name is &#123;{name}&#125;.
</div>
```

However, unlike Angular 1, this data binding is one way. By doing so, the amount of event listeners that were generated have been reduced, and thus, performance improved.

[ ]: Binding Properties

When a component needs to resolve and bind a variable, Angular now uses the [] syntax. We touched on this earlier in this chapter when covering about Inputs.

If we have this.currentColor in our component, we would pass this variable into our component, and Angular would ensure that the values would stay updated.

```
<card-header [themeColor]="currentColor"></card-header>
```

( ): Event Handling

In Angular 1, we would use custom directives to listen for user events, like clicking on an element (like ng-click). Angular 2 has taken a cleaner approach and just wraps the event you want to listen for in () and then assigns that to a function in the component.

```
<my-component (click)="onUserClick($event)"></my-component>
```

[( )]: Two-way data binding

By default, Angular no longer establishes two-way data binding. If you do need to have this functionality, the new syntax is actually a shorthand notation of the Binding property and the Event Handling syntaxes.

<input [(ngModel)]=”userName">

The `this.userName` value of your component will stay in sync with the input value.

***: The Asterisk**

The use of the asterisk before certain directives, tell Angular to treat our template in a special fashion. Instead of rendering the template as is, it will apply the Angular directive to it first. For example, `ngFor` takes our`<my-component>` and stamps it out for each `item` in `items`, but it never renders our initial `<my-component>` since it's a template:

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

Events

Events in Angular 2 use the parentheses notation in templates, and trigger methods in a component's class. For example, assume we have this template:

```
<button (click)="clicked()">Click</button>
```

And this component class:

```
@Component(...)
class MyComponent {
  clicked() {
  }
}
```

Our `clicked()` method in the component will be called when the button is clicked.

In addition, events in Angular 2 behave like normal DOM events. They can bubble up and propagate down.

If we need access to the event object, simply pass in the $event as a parameter in the event callback function.

```
<button (click)="clicked($event)"></button>
```

and the component class would become:

```
@Component(...)
class MyComponent {
  clicked(event) {
  }
}
```

Custom Events

What if you need to use some custom event that your components might need to interact with one another? Angular 2 makes this process quite easy.

In our component, we import the Output and EventEmitter modules. Then we define our new event, userUpdated, by using the @Output decorator. This event is an instance of an EventEmitter.

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

@Component({
  selector: 'user-profile',
  template: '<div>Hi, my name is </div>'
})
export class UserProfile {
  @Output() userDataUpdated = new EventEmitter();

  constructor() {
    // Update user
    // ...
    this.userDataUpdated.emit(this.user);
  }
}
```

When we want to trigger the broadcast of the event, you simply call the emit method on the custom event type and include any parameters to be transmitted with the event.

Now when we used this component elsewhere in our app, we can bind the event that `user-profile` emits

```
<user-profile (userDataUpdated)="userProfileUpdated($event)"></user-profile>
```

In a different component that imports our UserProfile component, can listen for the userProfileUpdated event that the component can broadcast.

```
import {Component} from '@angular/core';
import {UserProfile} from './user-profile';

export class SettingsPage {
  constructor(){}

  userProfileUpdated(user) {
    // Handle the event
  }
}
```

Lifecycle Events

Both the Angular app and its components offer lifecycle hooks that give developers access to each of the critical steps as they occur. These events are usually related to their creation, their rendering, and their destruction.

NgModule

The Angular team reworked the method of bootstrapping your application through the use of the NgModule function. This was done toward the end of the release candidate cycle for Angular, so it might come as a surprise to some. The @NgModule takes a metadata object that tells Angular how to compile and run module code. In addition, @NgModule allows you to declare all your dependencies up front, instead of having to declare them multiple times in an app.

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

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

export class AppModule { }
```

This code sample shows a basic aap.module.ts file that will use the BrowserModule to enable the Angular app to properly run in a browser, then both declare and bootstrap the AppComponent.

This module is in turned used by the main.ts file to perform the actual bootstrapping.

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

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModule);
```

This sample code initializes the platform that your application runs in, then uses the platform to bootstrap your AppModule. The Ionic starter templates will generate the necessary modules for you.

Another benefit of this system is it enables us to use the Ahead of Time (AoT) compiler, which provides for much faster applications.

Component Init Event

When a component is created, its constructor function is called. Within the constructor, any initialization we might need to perform on the component can occur. However, if our component is dependent on information or properties from a child component, we will not have access to that data.

Angular provides the ngOnInit event in order to handle to this need to wait. Our component can wait for this method to be triggered by the framework. Then all our properties are resolved and available to be used by the component.

Component Lifecycle Events

Beyond, ngOnInit, there are several other lifecycle events for a component:

ngOnDestroy

This method is called before the component is destroyed by the framework. This would be where you unsubscribe observables and detach event handlers to avoid memory leaks.

ngDoCheck

This method provides the ability to perform custom change detection.

ngOnChanges(changes)

This method is called when one of the component's bindings have changed during the checking cycle. The method's parameter will be an object in the format:

```
{
   'prop': PropertyUpdate
}
```

ngAfterContentInit()

Unlike ngOnInit, which is called before the content has been rendered, this method is called once that content is first rendered on the view.

ngAfterContentChecked

This method is called after Angular checks the bindings of the external content that it projected into its view.

ngAfterViewInit

After Angular creates the component's view(s) this method is triggered.

ngAfterViewChecked

The final method during the component initialization process. This method will be call once all the data bindings are resolved in the component's views.

Pipes

Pipes, previously known as "Filters," transform a value into a new value, like localizing a string or converting a floating point value into a currency representation:

```
<p>The author's birthday is &#123;{ birthday | date }&#125;</p>
```

If the birthday variable is a standard JavaScript Date object, it will look like `Thu Apr 18 1968 00:00:00 GMT-0700 (PDT)`. Certainly not the most human readable format. However, within the interpolation express in our template, our birthday value is passed through the pipe operator (|) to the Date pipe function on the right, thus rendering The author's birthday is April 18, 1968.

Angular comes with a stock set of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and`PercentPipe`. They are all immediately available for use in any template.

@ViewChild

Often we need to read or write child component values or call a child's component's method. When the parent component *class* requires that kind of access, we ***inject*** the child component into the parent as a *ViewChild*.

```
import {Component, ViewChild} from '@angular/core';
import {UserProfile} from '../user-profile';

@Component({
  template: '<user-profile (click)="update()"></user-profile>',
  directives: [UserProfile]
})

export class MasterPage {
  // we pass the Component we want to get
  // assign to a public property on our class
  // give it the type for our component
```

```
    @ViewChild(UserProfile) userProfile: UserProfile
    constructor() { }
    update(){
      this.userProfile.sendData();
    }
  }
```

Both the ViewChild module is injected from the Angular Core, as well as our UserProfile component. Within the Component decorator, we also must set the directives property to include a reference to our injected component. Our constructor contains our ViewChild decorator that set our userProfile variable to our injected component.

With those code elements in place, we then are able to interact with our child component's sendData method.

Understanding ES6 & TypeScript

Over the past few years, web developers have seen an explosion in attempts to create 'better' or more developer-centric versions of JavaScript. CoffeeScript, AtScript, Dart, ES6, TypeScript and so on have sought to improve on standard JavaScript. Each of these languages sought to extend JavaScript by providing features and functionality aimed at modern application development. But, each solution had to deal with the fact that our modern browsers use a version of JavaScript known formally as ECMAScript 5 (ES5). Meaning that each solution would need to output their efforts into standard JavaScript.

In order to use either of these modern language options, our code will have to be "transpiled" into ES5-based JavaScript. If you have never heard of the term transpiling before, it is a process of taking the code written in one language and converting into another.

Currently, there are two primary choices if you want to use 'next-generation' JavaScript: ES6 or TypeScript. ES6 is the next official version of JavaScript and was formally approved by the standards committee in June 2015, and over time it will be supported natively in our browsers. The other language option is TypeScript. Typescript is Microsoft's extension of JavaScript that comes with powerful type checking abilities and object-oriented features. It also leverages ES6 as part of its core foundation. TypeScript is a primary language for both Angular and Ionic application development.

Although none of our current browsers support either option, our code can be transpiled using tools like Babel or tsc. We don't have to worry about setting up this system, as it is built into the default Ionic build process.

Variables

With ES6, the way we can define our variables has improved. We can now specify a variable by using the keyword `let` .

In ES5, variables could only be defined using the keyword `var`, and they would be scoped to the nearest function. This was often problematic, as a variable could be accessible outside of the function that it was defined in.

```
for(var i in myList) {
  /*do something*/
}
```

The variable `i` is still available after the loop has finished. Not quite the expected behavior

By using the `let` keyword, this issue is no longer a problem. Let creates a variable that is scoped in the nearest block.

```
for(let i in myList) {
  /*do something*/
}
```

Now, after the loop has executed, `i` is not known to the rest of the code. Whenever possible, use let to define your variables.

Classes

JavaScript classes have been introduced in ES6 and are syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is **not** introducing a new object-oriented inheritance model to JavaScript. If you have developed using another object-oriented language like C# or Java, this new syntax should look very familiar. Here is an example:

```
class Rocket {
  landing(location) {
  }
}

class Falcon extends Rocket {
  constructor() {
    super();
    this.manufactuer = 'SpaceX';
    this.stages = 2;
  }
  landing(location) {
    if ((location == 'OCISLY') || (location == 'JRTI')){
      return 'On a barge';
    } else {
      return 'On land';
    }
  }
}

class Antares extends Rocket {
  constructor() {
    super();
```

```
        this.manufactuer = 'OrbitalATK';
        this.stages = 2;
      }
      landing(location) {
         console.log('In the ocean');
      }
    }
```

Promises

The **`Promise`** object is used for deferred and asynchronous computations. A `Promise` represents an operation that hasn't completed yet, but is expected in the future. This is exactly the type of functionality we need when interacting with remote servers or even loading local data. It provides a simpler method to handle async operations than traditional callback-based approaches.

A promise can be in one of 3 states:

- **Pending** - the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
- **Fulfilled** - the asynchronous operation has completed, and the promise has a value.
- **Rejected** - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.

The primary API for a promise is its `then` method, which registers callbacks to receive either the eventual value or the reason why the promise cannot be fulfilled.

Assuming we have a function sayHello that is asynchronous and needs to look up the current greeting from a web service based on the user's geolocation, it may return a promise:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'Hello in the United States'
});
```

The advantage of this method is, while the function is awaiting the response from the server, the rest of our code can still function.

In case, something goes wrong, like if the network goes down and the greeting can't be fetched from the web service, you can register to handle the failure using the second argument to the promise's `then` method:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'Hello in the United States'
}, function (error) {
```

```
    console.error('uh oh: ', error);   // 'Drat!'
});
```

If `sayHello` succeeds, the greeting will be logged, but if it fails, then the reason, i.e. error, will be logged using `console.error`.

Observables

Many services with Angular use Observables instead of Promises. Observables are implemented through the use of the RxJS library (https://github.com/ReactiveX/RxJS). Unlike a Promise which resolves to a single value asynchronously, an observable resolves to (or emits) multiple values asynchronously (over time).

In addition, Observables are cancellable and can be retried using one of the retry operators provided by the API, such as retry and retryWhen. Promises require the caller to have access to the original function that returned the promise in order to have a retry capability.

Template Strings

One of the features of Angular is its built-in templating engine. In many cases, these templates are stored as external files. However, there are times when keeping them inline makes more sense. The difficulty has been writing long inline without having to resort to using concatenation or needing to escape any single or double quotes in the string.

ES6 now supports the use of backticks at the start and end of the string.

```
let template = `
  <div>
    <h2>&#123;{book.name}&#125;</h2>
    <p>
      &#123;{book.summary}&#125;
    </p>
  </div>
`;
```

Template strings do not have to remain static. You can perform string interpolation by using $(expression) placeholders:

```
let user = {name:'Rey'};
let template = `
  <div>Hello, <span>${ user.name }</span></div>
`;
```

Template Expressions: ES6 vs. Angular's

ES6's template expression are only meant for string replacemen. If you need to evaluate a function or test a condition, use Angular template expressions instead.

Arrow Functions

Arrow functions make our code more concise, and simplify function scoping and the `this` keyword. By using arrow function we avoid having to type the `function`keyword, `return` keyword (it's implicit in arrow functions), and curly brackets.

In ES5, we would have written a simple multiply function like this:

```
var multiply = function(x, y) {
  return x * y;
};
```

but in ES6 using the new arrow function formation, we can write thesame function this way:

```
var multiply = (x, y) => { return x * y };
```

The arrow function example above allows us to accomplish the same result with fewer lines of code and approximately half of the typing.

One common use case for arrow functions is array manipulations. Take this simple array of objects:

```
var missions = [
  { name:'Mercury', flights:6 },
  { name:'Gemini', flights:10 },
  { name:'Apollo', flights:11 },
  { name:'ASTP', flights:1 },
  { name:'Skyalb', flights:3 },
  { name:'Shuttle', flights:135 },
  { name:'Orion', flights: 0 }
];
```

We could create an array of objects with just the names or flights by doing this in ES5:

```
// ES5
console.log(missions.map(
  function(mission) {
    return mission.flights;
  }
)); // [6, 10, 11, 1, 3, 135, 0]
```

Rewriting this using arrow function, our code is more concise and easier to read:

```
// ES6
console.log(missions.map(
  mission=>mission.flights
)); // [6, 10, 11, 1, 3, 135, 0]
```

Types

TypeScript is a data-typed language, which gives you compile-time checking. By default, TypeScript supports JavaScript primitives: **string**, **number,** and **boolean**.

```
let num: number;
let str: string;
let bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

TypeScript also supports typed **arrays**. The syntax is basically postfixing [] to any valid type annotation (e.g. :boolean[]).

```
let booleanArray: boolean[];

booleanArray = [true, false];
console.log(booleanArray[0]); // true
console.log(booleanArray.length); // 2
booleanArray[1] = true;
booleanArray = [false, false];

booleanArray[0] = 'false'; // Error!
booleanArray = 'false'; // Error!
booleanArray = [true, 'false']; // Error!
```

Special Types

Beyond the primitive types, there are few types that have special meaning in TypeScript. These are **any**, **null**, **undefined**, and **void**.

```
let someVar: any;

// Takes any and all variable types
someVar = '123';
someVar = 123;
```

The `null` and `undefined` JavaScript literals are effectively treated as the same as the `any` type.

```
var str: string;
var num: number;

// These literals can be assigned to anything
str = undefined;
num = null;
```

Typing Functions

Not only can you type variables, but you can also type the results of a function call.

```
function sayHello(theName: string): string {
    return 'Hello, '+theName;
}
```

If you try to call sayHello and assign the result to an incorrect data type, the compiler will throw an error.

:void

Use :void to signify that a function does not have a return type.

```
function log(message): void {
    console.log(message);
}
```

Summary

There is so much more to cover in both Angular and TypeScript than in this chapter's very simple introduction. In all likelihood, you will be using additional resources for both of these technologies as you begin to develop more feature-complete applications. But let's look at some of the key points we have covered in this chapter:

We looked at Angular's new component model, its templating, the new data-binding methods, and the component life-cycle. Beyond that, we also explored some new capabilities in ES6 and TypeScript. These included: Classes, Promises, Arrow Functions, as well as the ability to assign types to our variables and functions.

CHAPTER 6
Apache Cordova Basics

The Ionic Framework is built atop several technologies; Angular and Apache Cordova. In this chapter, we will explore what Apache Cordova is, and how it interacts with Ionic.

Apache Cordova is an open source framework that enables mobile app developers to take their HTML, CSS, and JavaScript content and create a native application for a variety of mobile devices. Let's look further at how this works.

Cordova takes your web application and renders it within a native web view. A web view is a native application component (like a button or a tab bar), that is used to display web content within a native application. You can think of a web view as a web browser without any of the standard user interface chrome, like a URL field, or status bar. The web application running inside this container is just like any other web application that would run within a mobile browser. It can open additional HTML pages, execute JavaScript code, play media files, and communicate with remote servers. This type of mobile application is often called a hybrid application.

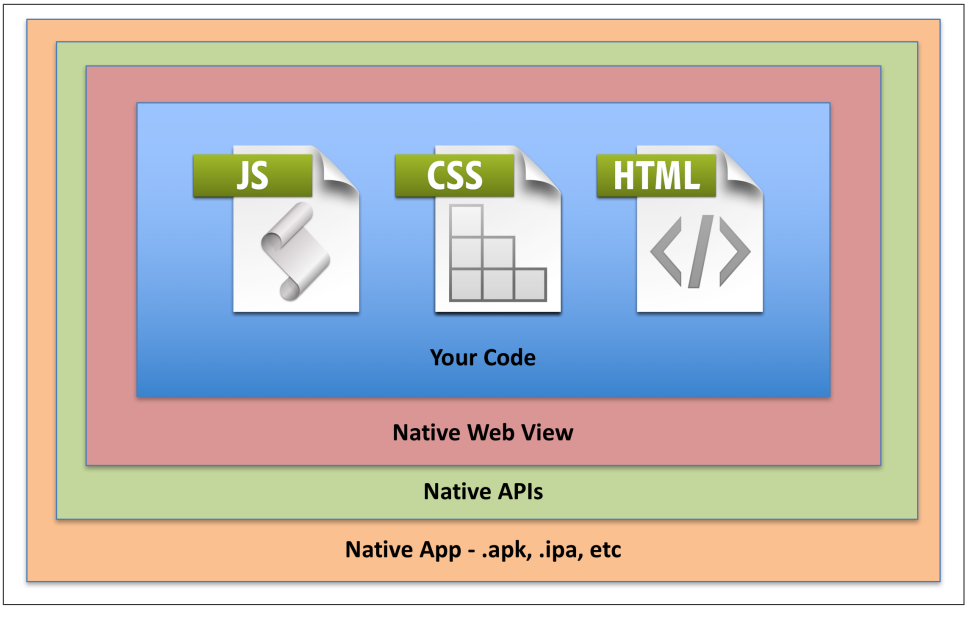


Figure 6-1. How Cordova applications are composited to create fully native applications

Typically, web-based applications are executed within a sandbox. Meaning that they do not have direct access to various hardware and software features on the device. A good example of this is the contact database that exists on your mobile device. This database of names, phone numbers, emails and other bits of information is not accessible to a web app. Besides providing a basic framework to run a web app within a native application, Cordova also provides JavaScript APIs to allow access to a wide variety of device features, like the contacts database. These capabilities are exposed through the use of a collection of plugins. We will explore their use later in this book, but plugins provide a bridge between our web application and the device's native features. There is a core set of plugins that is maintained by the Cordova project, as well as a large collection of third party plugins that offer even more functionality (NFC communication, Force Touch, Push Notifications, just to name a very few).

Table 6-1. Core Plugins

Battery Status	Monitor the status of the device's battery.
Camera	Capture a photo using the device's camera.
Console	Provides an improved console log
Contacts	Work with the device's contact database.
Device	Gather device specific information.
Device Motion (Accelerometer)	Tap into the device's motion sensor.

Device Orientation (Compass)	Obtain the direction that the device is pointing.
Dialogs	Visual device notifications.
File	Hook into native file system through JavaScript.
File Transfer	Allows your application to upload and download files
Geolocation	Make your application location aware.
Globalization	Enable representation of objects specific to a locale.
InAppBrowser	Launch URLs in another in-app browser instance.
Media	Record and play back audio files.
Media Capture	Captures media files using device's media capture applications.
Network Information (Connection)	Quickly check the network state and cellular network information.
SplashScreen	Show and hide the application's splash screen.
StatusBar	An API for showing, hiding and configuring status bar background.
Vibration	An API to vibrate the device.
Whitelist	Implements a whitelist policy for navigating the application webview.

The History of Cordova (aka PhoneGap)

Developers are often confused by the difference between Apache Cordova and PhoneGap. In an attempt to clear up this confusion, we need to understand the origins of this project. In late 2008, several engineers from Nitobi attended an iPhone Dev Camp at the Adobe offices in San Francisco. They explored the idea of using the native web view as a shell to run their web applications in a native environment. The experiment worked. Over the next few months, they expanded their efforts and were able to leverage this solution to create a framework. They named the project, PhoneGap, since it allowed web developers the ability to bridge the gap between their web apps and the device's native capabilities. The project continued to mature, more plugins were created, enabling more functionality to be accessed on the phone. Other contributors joined the effort, expanding the number of mobile platforms it supported.

In 2011, Adobe bought Nitobi and the PhoneGap framework was donated to the Apache Foundation. The project was eventually renamed Cordova (which is actually the street name of Nitobi's office in Vancouver, Canada).

Apache Cordova vs. Adobe PhoneGap

Since there is both Apache Cordova and Adobe PhoneGap, it is quite easy to become confused between the projects. This naming issue can be the source of frustration when researching an issue during development and having to search using both Cordova and PhoneGap as keywords to find the solutions. Or even in reading the proper documentation, as the two projects are so intertwined.

A good way to try to understand the two projects is to think about how Apple has its Safari browser, but it is based on the Open Source WebKit engine. The same is true here, Cordova is the Open Source version of the framework; while PhoneGap is the Adobe-branded version. But in the end, there is little difference between the two efforts at this time. There are some slight differences in the command line interfaces between Cordova and PhoneGap, but again the functionality is the same. The only thing you cannot do is mix the two projects in the same application. While not as dangerous as when the Ghostbuster's crossed their streams, using both Cordova and PhoneGap in the same project will produce nothing but trouble.

A personal note

We tend to use PhoneGap as our primary search term since that was its original name when researching issues.

The main difference between the projects is that Adobe has some paid services under the PhoneGap 'brand', most notably the PhoneGap Build service (https://build.phonegap.com/). This is a hosted service that enabled you to have your application compiled into native binaries remotely, eliminating the need to install each mobile platform's SDKs locally. The PhoneGap CLI has the ability utilize this service, while the Cordova CLI does not.

The other difference is the PhoneGap CLI can be used in tandem with the PhoneGap Developer App. This free mobile app (http://app.phonegap.com/) allows for your app to run on-device without the need to first compile it. This provides a very easy method to test and debug your application on-device. Don't worry there is an Ionic equivalent for us to leverage (http://view.ionic.io/), and we will be using it during our development cycle.

In this book, we will be using Cordova command-line tool when we are not using the Ionic command-line tool. This is in part due to the fact the Ionic CLI is based on the Cordova CLI and not the PhoneGap CLI.

A Deep Dive into Cordova

In the past configuring a Cordova project was a difficult task. It meant first creating a native application project in each platform's IDE, then modifying it support the interfaces for Cordova. With the release of the command line tool, this process became easier. The CLI scaffolds a basic project and configures it to work with any supported mobile platform you can use. The Cordova CLI, also allows us to have easy integration and management of the plugins as well for our project. Finally, the CLI enables us to quickly compile our app to run in a simulator or on an actual device. We will be exploring these commands further as we work through our sample applications.

Configuring your Cordova app

Each Cordova application is defined by its config.xml file. This global configuration file controls many of the aspects of the application from app icons, plugins, and range of platform-specific settings. It is based on the W3C's Packaged Web Apps (Widgets) specification (http://www.w3.org/TR/widgets/). The Ionic CLI will generate a starter config.xml file during its scaffolding process.

As you develop your Ionic application, you will need to update this file with new elements. Some common examples of this are adjusting the app name, the app id string, or adding a platform-specific setting like android-minSdkVersion.

A deeper exploration of the config.xml can be found in Appendix A.

Device Access (aka Plugins)

Some of the real power of Cordova comes from its extensive plugin library. At the time of this writing, there were over 1250 plugins listed at http://cordova.apache.org/plugins/#/. But what is a Cordova plugin? From the site, they define a plugin as:

> A plugin is a bit of add-on code that provides JavaScript interface to native components. They allow your app to use native device capabilities beyond what is available to pure web apps.

As mentioned earlier, there are two sets of plugins: core and third party. Time for another brief history lesson, up to version 3.0 of PhoneGap (pre-Cordova), the code base contained both the code to use the webview and its communication to the native app, along with some of the 'key' device plugins. Starting with version 3.0, all the plugins were separated out as individual elements. That change means that each plugin can be updated as needed, without having to update the entire code base. These initial plugins are known as the 'core' plugins.

But what about the other 1220 or so plugins, what do they provide? An incredibly wide range of things: Bluetooth connectivity, push notifications, TouchID, 3D Touch, just to name a few.

Interface Components: The missing piece

While Cordova provides a lot of functionality to the developer in terms of being able to leverage their code across multiple platforms, the ability to extend beyond the web and use native device features, it is missing one critical component: user interface elements. Beyond just the basic controls that HTML provides by default, Cordova does not offer any components like those found in a native SDK. If you want to have a tab bar component you are either going to have to create the HTML structure, write the needed CSS, and develop the JavaScript to manage the views or use a third party framework, like Ionic.

This lack of a UI layer has often been one of the difficulties in working with Cordova. With the release of the Ionic Framework, developers now have a first-rate interface toolkit with which they can author their mobile applications. There are other solu-

tions available for a Cordova application, but this book is focused on Ionic. If you would like to look at some other options, you might look at OnsenUI, Framework7 or ReactJS as an option. For me, I have been very pleased with what I have been able to build Ionic and their services.

Why Not Cordova?

Although Cordova is an incredibly powerful solution for building mobile applications, it is not always the right choice. Understanding the advantages and disadvantages of the framework is critical in how well your application will perform. Our Cordova application is several layers away from the actual native layer. So, great care must be taken to develop a performant Cordova application. This was very true during the early days of PhoneGap when devices were relatively low powered. Current mobile devices have much more computing power that can be used.

With that said, you are not going to develop the next Angry Birds or Temple Run using Cordova. But you might be able to build the next Quiz Up, Untappd or Trivia Crack.

Understanding Web Standards

Another thing to consider is what the webview is actually capable of. Although usually, mobile webviews are fairly current with recognized web standards, it is always worthwhile to understand there might be issues or quirks that you need to be aware of. If you have been doing any modern web development, there is a good chance you have used caniuse.com.

This is an excellent resource to check if a particular HTML5 feature is available, or if a CSS property is supported. For example, if we wanted to use scalable vector graphics (SVGs) in our application to assist in dealing with a range of screen sizes and densities, we can go caniuse.com can see when each mobile platform began supporting it.

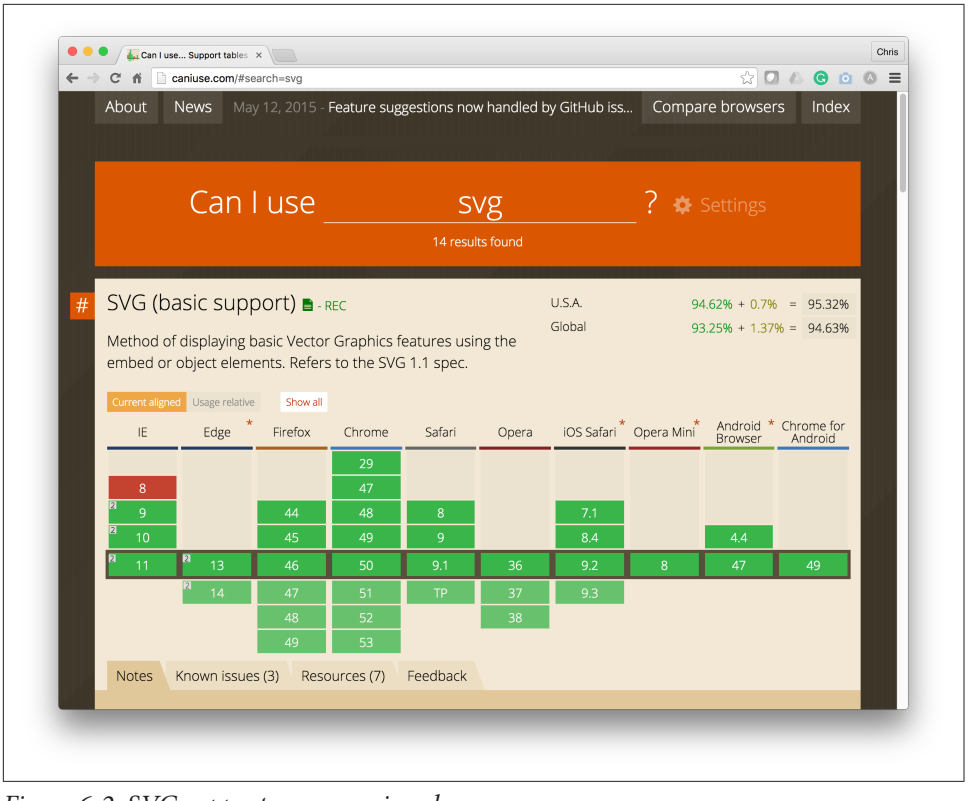


Figure 6-2. SVG support across various browsers.

In this case, we can see that while iOS has had support for several versions, the Android platform only recently enabled support for this format.

It is worth noting exactly where the webview that Cordova uses comes from. When you compile your application, Cordova does not actually include a webview into the application. Instead, it will use the webview that is included on the device as part of it's OS. Meaning for current iOS devices, you get a version of WebKit, while certain Android device will be using Chromium, while others might still be using WebKit.

Recently the Intel Open Source Technology Center released their Crosswalk Project (https://crosswalk-project.org/). This is an effort replace the default webview in your Cordova project with the latest version of Google Chromium, thus giving you the latest browser support. Although using this solution will increase the size of your application, it can help in normalizing the web features that you can use in your application. If your application will be used by older Android devices, using Crosswalk can great resolve some of the browser issues that exist.

Summary

Let's summarize what we covered in this chapter:

- Apache Cordova was originally known as PhoneGap. PhoneGap is now the Adobe-branded version
- Device capabilities are added through the use of a plugin system
- Cordova can be used for a wide range of mobile applications, but may not always be the right technology choice
- Cordova enables you to use your web technology skills to create mobile applications for a variety of mobile platforms
- Cordova does not provide a user interface library, so an additional framework will be needed (like Ionic)

CHAPTER 7
Understanding Ionic

Let's now take a deeper look at what makes up the foundation of an Ionic page. Each Ionic page is formed from three base files: an HTML file which defines the components that are displayed, the Sass file which defines any custom visual styling for those components, and the TypeScript file which will provide the custom functions for those components. Since Ionic is built upon web technologies, we can use many of the same solutions that we might use in developing a traditional web application or web page.

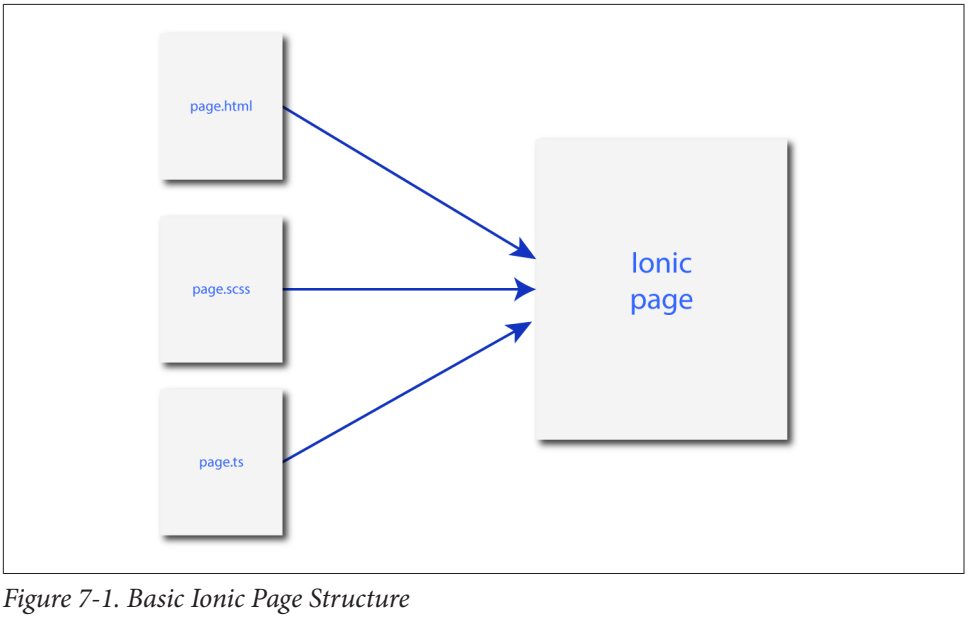


Figure 7-1. Basic Ionic Page Structure

HTML Structure

Unlike a traditional HTML file, you do not need to include the `<head>` tag and any elements you normally would include; like importing CSS files or other code libraries. You also do not include the `<body>` tag nor even the `<!DOCTYPE html>` or `<html lang="en" dir="ltr">` tags. The contents of this HTML file are rendered within our application container, so we do not need them. We just define the actual components that will be shown to the user. These components are a mixture of traditional HTML tags, as well as, custom tags that are used to define the Ionic components. Here is a sample of an Ionic page and how it is rendered:

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

<ion-content padding>
  The world is your oyster.
  <p>
    If you get lost, the <a href="http://ionicframework.com/docs/v2">↵
    docs</a> will be your guide.
  </p>
</ion-content>
```

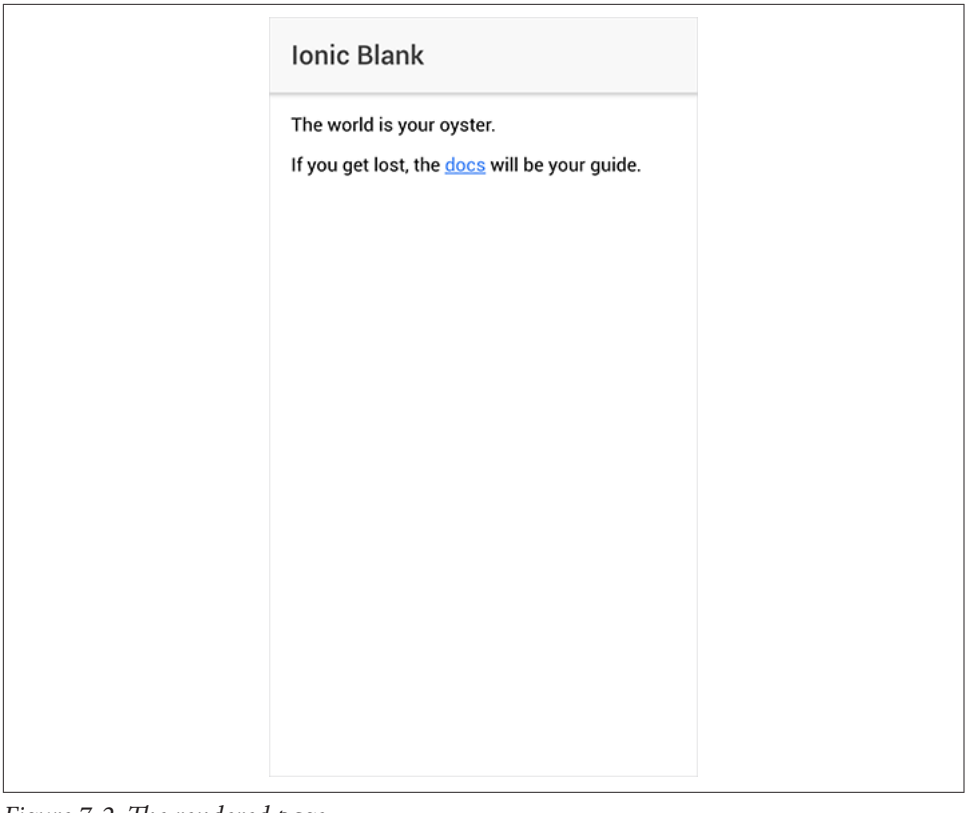


Figure 7-2. The rendered page.

You can see the markup is a blend of standard HTMLs (`<p>` and `<a>`) and Ionic tags (`<ion-header>`, `<ion-content>`, etc).

Ionic Components

One of Angular's features is the ability to extend the vocabulary of HTML to include custom tags. The Ionic framework leverages this capability and has created an entire set of mobile components. These include components like `<ion-card>`, `<ion-item-sliding>` `<ion-segment-button>` and more. For a complete summary of the Ionic component library, see Appendix B as well as the Ionic documentation (http://ionic-framework.com/docs/v2/). All of the Ionic's components are pre-fixed with the **ion-**, so they are easy to spot in the markup.

Autocompleting Ionic Tags

Most code editors offer some form of autocomplete or code-hinting that can be extended. There is usually a downloadable plugin for your editor that enables code hinting for the Ionic framework.

Behind the scenes, each of the components is decomposed at runtime into basic HTML tags with their custom CSS styling and functionality.

Just like standard HTML tags, Ionic components also accept various attributes to define their settings, such as setting their id value or defining additional CSS classes. But Ionic components are also extended through additional attributes like

```
<ion-item-options side="left">
```

or using Angular directives:

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

As we build our sample applications you will become more familiar with the Ionic component library.

Understanding the SCSS file

The visual definition of an Ionic application is defined by CSS. Ho wever, this CSS is actually generated by Sass or Syntactically Awesome Style Sheets. If you have never worked with Sass, it provides several advantages over writing CSS directly. These include the ability to declare variables like **$company-brand: #ff11dd**. You can reference this variable instead of directly assigning the color. Now if we had to change our color, we only have to do it in one location and not across multiple files.

CSS Variable Naming

It is often tempting to give the variable a name like **$company-red**, but consider what happens if the branding changes and instead of red, the color is actually green?

All of Ionic's components are styled using Sass variables. For example, we can change the **$list-background-color** by adding our value in the *app.variables.scss* file. The team has done an incredible job in ensuring that each Ionic component is easy to style. Refer to the Ionic documentation for a complete list of all the configurable Sass variables.

Sass also supports an improved syntax for writing nested CSS. Here is an example of this:

```
nav {
  ul {
```

```
      margin: 0;
      padding: 0;
      list-style: none;
    }

    li { display: inline-block; }

    a {
      display: block;
      padding: 6px 12px;
      text-decoration: none;
    }
  }
```

which when transformed becomes:

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

This method of writing CSS shorthand can be a real time saver.

Typically the screen's .scss file is where you will define any page specific CSS. For example, if you need to have a slightly different button style for a login screen. By defining it in the associated .scss file, you keep the associated elements package together in a more logical fashion. For any application-wide theming, that styling should be defined in the *app.core.scss* files.

We will explore theming in greater detail when we are building our sample applications.

Understanding TypeScript

The last element needs to create an Ionic screen is the associated TypeScript file (or .ts file). This where all the Angular/TypeScript code that we will write to control the interactions of this page will be written.

The file will define any code modules we need to import in order for our screen to function. Typically this would be components that we might need to programmatically interact with (like navigating to a new screen) or Angular modules that offer

needed functions (like making an HTTP request). It will also define the base component that is actually our screen. As we need to add functions to respond to user input, that code will also be added into this file. Here is what a basic .ts file looks like:

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

@Component({
  templateUrl: 'build/pages/home/home.html'
})
export class HomePage {
  constructor(public navCtrl: NavController) {

  }
}
```

This TypeScript code just defines the HomePage component and links it the template.

We will be working a lot of TypeScript throughout our sample applications, so we will not go any further in exploring this element.

Conclusion

You should have a better understanding of the three key elements that make up a single Ionic screen: the HTML file, the Sass file, and TS file. Let's move on to building our first actual Ionic application.

CHAPTER 8
Building our Ionic 2 Do App

With our development environment configured, some initial exposure to Angular 2, and a foundation in Apache Cordova, we are finally ready to start creating our first Ionic 2 application. As not to break with tradition, we are going to be building the classic To Do list management application. You might wonder why would build something that has been built so many times before. Part of the reason is for many of you building something familiar will let you begin to map how Ionic works to whatever language or framework you might be more familiar with. Another reason is that a To Do app has more complexity that simply printing out "Hello World" on a screen.

To get started we need to create a new Ionic project. We will use the blank template as our basis.

```
$ ionic start Ionic2Do blank --v2
```

Make sure you include the **--v2** flag, otherwise you will create an Ionic version 1 application. The --v2 flag will tell the Ionic CLI that you want a version 2 based project.

The CLI will begin downloading the various elements for the project; the TypeScript components, the Node modules, and finally the required Cordova components. This process may take a few minutes depending on your internet connection. If you have done Ionic v1 development, you will notice that this process is a bit longer than before.

Once all the packages have been downloaded, the CLI will happily inform you that your Ionic app is ready to go!

Next, you need to make sure you change the working directory into your newly created Ionic project's directory.

```
$ cd Ionic2Do
```

More than once I have forgotten this simple step and wondered why my next Ionic command would fail. The CLI itself provides a gentle reminder when it finishes as well.

Adding our Platforms

If you are building on a Mac, the Ionic CLI will automatically include the iOS platform for us. Since building for that platform is not an option in Windows, the CLI will not attempt to add it. Since Android can be built on either a Mac or PC, we will add that platform as well to our project.

```
$ ionic platform add android
```

This will add a platforms directory to our project, as well as the platforms.json file. The platforms directory will have sub-directories of each of the platforms that we include. If you look at the platforms > android directory, you will see all the platform specific elements needed to build our application for the Android OS; the Gradle files, the AndroidManifest.xml, and so on.

To add the iOS platform, we just need to change the platform name in CLI command

```
$ ionic platform add ios
```

And now the platforms directory will now contain an iOS directory, that contains the iOS specific elements, like the Xcode project file.

Previewing our Ionic2Do App

Let's go ahead and take a quick look at what the Ionic2 blank template looks like in our browser. Although there is an index.html file located inside the www directory, we actually can not open it directly in our browser. Instead, this file needs to be served from an actual web server. Luckily, we do not need to upload this to an actual server, the Ionic CLI has a command that will spin up a local web server, and allow us to preview our app.

```
$ ionic serve
```

The Ionic CLI will begin the process of compiling and bundling the dependencies. Then it will compile the Sass files into CSS, insert the IonIcon font library, and finally, the HTML files are copied. A web server is started, and Google Chrome is launched (or if already running, a new tab will be created) and our Ionic app is displayed

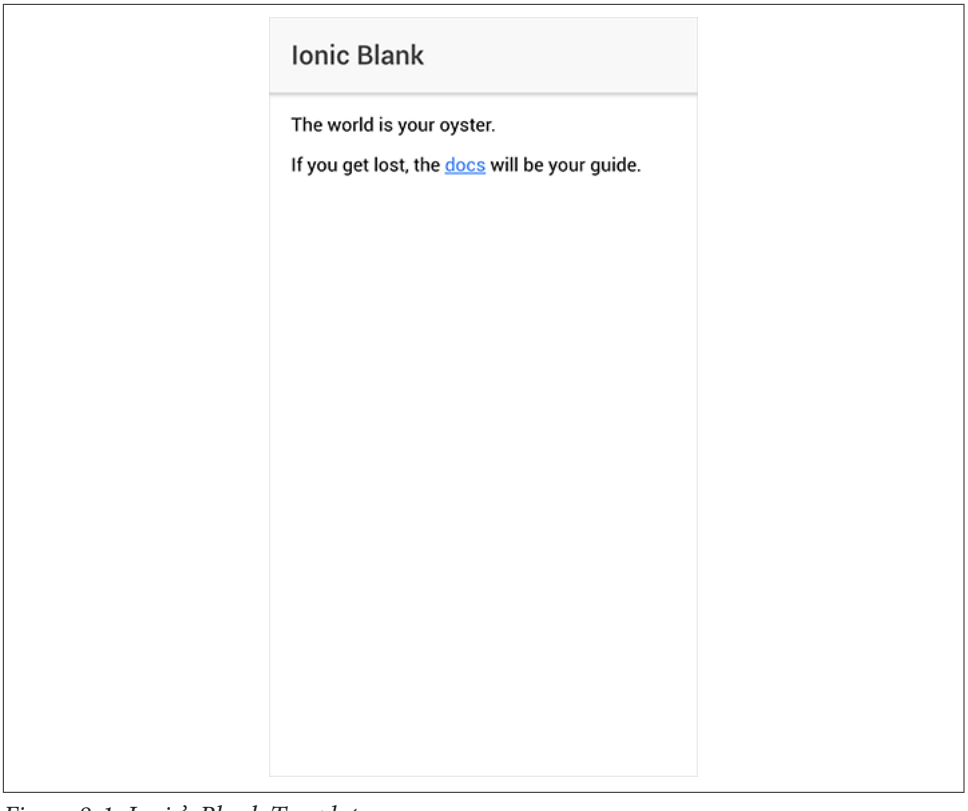


Figure 8-1. Ionic's Blank Template

The terminal window displays a list of some of the server commands.

Command	Action
restart or r	restart the client app from the root
goto or g and a url	to have the app navigate to the given url
consolelogs or c	enable/disable console log output
serverlogs or s	enable/disable server log output
quit or q	shutdown the server and exit

The server will watch the project files for changes and reload the app once you have saved a file.

In addition, you can start the server with the console logging enabled by adding a **--c** flag, as well as enabling the server logs with the **--s** flag

But the flag I usually enable is the **--lab** flag. This flag will tell the server to create three copies of our Ionic app in the browser. One will have the platform flag set to iOS, another set to Android, and the third set to Windows. This will allow you to quickly preview the differences in the UI between the three platforms, and a simpler way to test any CSS changes you might be applying to your project.

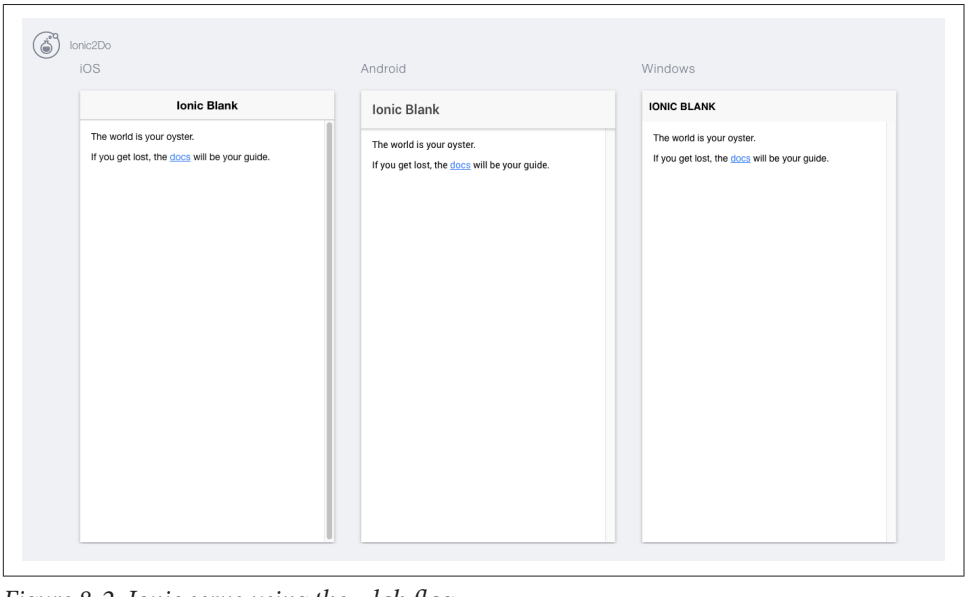


Figure 8-2. Ionic serve using the --lab flag.

To quit the server, just type in **q** into the terminal window, and press the enter key.

Previewing a specific platform

When using just `$ ionic serve` , it will use android as its platform of choice. To see your application as it would be rendered on different, just append **?ionicplatform=platformname** to end of the url. The values are **ios**, **android**, and **windows**.

Let's return back to the project directory and look at some additional files and how they are related.

Exploring the project structure: the index.html file

The first file I want to look at is the *www/index.html*. With Ionic 1, there was actually a lot of content that was placed in this file. Here is a portion of one of my Ionic 1 app's script tags

```
<!-- ionic/angularjs js -->
<script src="lib/ionic/js/ionic.bundle.js"></script>
<script src="lib/ionic-service-core/ionic-core.js"></script>
<script src="lib/ionic-service-analytics/ionic-analytics.js"></script>
<!-- ngCordova -->
<script src="lib/ngCordova/dist/ng-cordova.min.js"></script>
<!-- cordova script (this will be a 404 during development) -->
<script src="cordova.js"></script>
<!-- App js -->
<script src="js/app.js"></script>
<!--Factories/Services-->
<script src="js/hikedata.js"></script>
<script src="js/mapdata.js"></script>
<script src="js/appStatus.js"></script>
<script src="js/geoLocService.js"></script>
<!--Controllers-->
<script src="views/home/home.js"></script>
<script src="views/hikes/hike.js"></script>
<script src="views/hikelist/hikelist.js"></script>
<script src="views/hikedetails/hikedetails.js"></script>
<script src="views/map/map.js"></script>
<script src="views/about/about.js"></script>
```

As you can see there are a whole lot of script tags that were added into the index.html file during development. With Ionic 2, all of this JavaScript loading management is baked into the CLI and the build process. We now just have three script tags. Certainly a much more manageable list. In fact, we no longer have to remember to add script tags for every Angular element, the build process now generates a single *main.js* file with all our code merged together.

```
<!-- cordova.js required for cordova apps -->
<script src="cordova.js"></script>

<!-- The polyfills js is generated during the build process -->
<script src="build/polyfills.js"></script>

<!-- The bundle js is generated during the build process -->
<script src="build/main.js"></script>
```

If we look at the actual content within the `<body>` tag, we will just find one tag, `<ion-app>`. Through the use of this one component, our entire app will be bootstrapped

Another change from Ionic 1 to Ionic 2, is the CSS is generated from the Sass files by default. With Ionic 1, you had to enable this process. Now in Ionic 2, everything is built from the Sass files into a single CSS file.

```
<link href="build/main.css" rel="stylesheet">
```

With the shift to a component focused development paradigm, almost all the real code is now within the App component. Let's explore the new src directory in more detail.

Within the src directory, we find four directories: app, assets, pages, and theme. We also find four files, a *declarations.d.ts* file, an *index.html* file, a *manifest.json* file, and a *service-worker.js* file.

Exploring the project structure: the app directory

During the beta releases of Ionic 2, there was an app.ts file that contained the general bootstrapping Angular code for the application. With the shift to NgModule, and Angular itself stabilizing its recommended directory structures, the Ionic directory structure followed suit. Now, the initial app files are stored within the app directory, while the rest of the app is located in the pages directory.

Inside the app directory, we will find five files. Here is a brief summary of them:

app.component.ts	This file contains the base component that our app will initially use.
app.module.ts	This file declares the initial modules, providers, and entry components.
app.scss	This file defines any global CSS styles.
main.dev.ts	This TypeScript file is used during development to load our app.
main.prod.ts	This TypeScript file is used for the production release of our app.

Let's explore each of these files in a bit more depth to understand their role. First, open *app.module.ts* in your editor.

Visual Studio Code

The relationship between a developer and their code editor is a special one. In working with Ionic 2, I personally have found using Microsoft's Visual Studio Code as my editor of choice. It is a free download from https://code.visualstudio.com/. It obviously has support for TypeScript, as well as Cordova and Ionic.

This file declares what the NgModule function will do when called as Angular/Ionic runs.

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { HomePage } from '../pages/home/home';

@NgModule({
  declarations: [
    MyApp,
    HomePage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
```

```
    ],
    bootstrap: [IonicApp],
    entryComponents: [
      MyApp,
      HomePage
    ],
    providers: []
  })
  export class AppModule {}
```

There are several Ionic specific things to note here. First, importing IonicApp and IonicModule from the ionic-angular library. The IonicModule's forRoot method is called to define the root app component. We can pass a config object into IonicModule. This object allows us to define such items like; the back button text, the icon mode, or event the tab placement. Since we are not changing any of these settings, we will leave them as their platform defaults. For a complete list of all the configuration settings, see http://ionicframework.com/docs/v2/api/config/Config/.

We also define NgModule's bootstrap parameter to use IonicApp for the actual bootstrap process.

Second, the importing the main app component, in this case, MyApp and the first view, HomePage. These are both included in the entryComponents array to be used by the Ahead of Time (AoT) compiler.

Now, let's explore the *app.component.ts* file in more depth. At the start of the file are the import statements. If you recall, this is how Angular 2 handles its dependency injection.

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

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

The first import statement loads the Component module from Angular core.

What about the @App Component?

In earlier versions of Ionic 2, the application was bootstrapped using the App module. With the release of beta 8, the Ionic team shifted away from this custom module to use the generic Component module instead.

The next import statement loads the Platform module from the Ionic framework libraries. The third import statement loads the StatusBar plugin from the Ionic Native library. The final import statement loads our HomePage component that is defined in the home directory inside of the pages directory. We will touch on this module in a bit.

Next, comes the actual @Component decorator.

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

What is a decorator?

Decorators are simply functions that modify a class, property, method, or method parameter.

Within our Component decorator, we will specify the template to render. The template can be declared in one of two forms; inline like our example, or as an external file and referenced using templateUrl instead.

Since the template is short, we can just include it inline. For more complex HTML templates, you will want to keep them in a separate file.

Our template is just the `<ion-nav>` component. Just as in Ionic 1, this is basic navigation container for our content. We will look at the Navigation components in a bit. Also included in our template is set the root page for the component. We use Angular2's new one-way data-binding syntax of [ ] to set the root property to the rootPage variable.

The Ionic config options are left in the default state by passing in an empty object. We will explore these options later, but this is where we can set globally, things like back button styles, animation effects, icon types and more. This a nice change from Ionic 1, where these parameters were scattered throughout the framework.

```
export class MyApp {
  rootPage: any = HomePage;

  constructor(platform: Platform) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      StatusBar.styleDefault();
    });
  }
}
```

The Ionic CLI auto names this class MyApp. Next, it defines the rootPage variable to be the HomePage component. Since this is TypeScript, we also define the variable type, in this case, we are setting it to be of type 'any'.

The component's constructor has the Ionic Platform component passed as a parameter. Within the constructor, a JavaScript promise is made from the platform.ready

function. Once Ionic and Cordova are done bootstrapping, this promise will be returned and any device specific code will be safe to execute.

DeviceReady Event

If you have done any Cordova or PhoneGap development in the past, then you are probably familiar with the deviceReady event that Cordova will fire once the web-to-native bridge has finished initializing. It is similar to the documentReady event that traditional web development listens for.

In earlier versions of Ionic 2, we would have made a call to ionicBootstrap, but this is now handled within NgModule function.

The next file within our app directory is the *app.scss* file. This file serves as a global Sass file for our app. If you are not familiar with Sass, it is a stylesheet language that compiles to CSS. We will explore styling and theming later. For now, we can leave this file as is.

The final two files, *main.dev.ts* and *main.prod.ts* are entry points to our application, depending on the mode our app is running in development or production. Here is what is in the *main.dev.ts* looks like:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

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

The platformBrowserDynamic module is loaded, as is our AppModule. Then the bootstrapModule function is called and our AppModule passed in. For the production version of *main.ts*, the enableProdMode module is loaded and called before we make the bootstrap call.

Remember, these files are about the initial start of the application. You should not need to interact with these files much except to override a configuration or a global styling change.

The Pages Directory

The Ionic CLI will also generate a Pages directory. With Ionic 1, it was common to keep the HTML templates in one directory and the related *controller.js* file in another directory. Now, it is considered best practice to keep all the associated files together. So, within the pages directory, we will find a home directory. Inside this folder, we will find *home.html*, *home.scss*, and *home.ts*. As you build out your Ionic 2 application, you will create new directories for each page and place them within the pages directory.

Exploring the project structure: the home.html file

Looking at the HTML code of this file, we will see that all the tags begin with <ion-. One of the original reasons the Ionic Framework was built atop the Angular Framework, was the ability to extend the HTML language to include markup that represented the mobile components that were needed. This is still true with Ionic 2.

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

<ion-content padding>
  The world is your oyster.
  <p>
    If you get lost, the <a href="http://ionicframework.com/docs/v2">
    docs</a> will be your guide.
  </p>
</ion-content>
```

In order for our `<ion-navbar>` to be above the content and remain fixed during page transitions, it must be placed within a `<ion-header>` tag. Now within the `<ion-navbar>`, we find the `<ion-title>` tag that sets the title of the Navbar. This component is platform-aware. On iOS, the text is centered within the Navbar, as for the Apple iOS Human Interface guidelines, but on Android or WindowsPhone the text is aligned left. This is a great example of defining the base component, and Ionic doing the heavy lifting of applying the platform specific styling.

Beneath the navbar, is the `<ion-content>`. This defines the main container that we will place most of our app's interface. It has one Angular directive, padding, to provide some CSS padding to the container. The blank template has some text, a <p> tag, as well as, a <a> tag. This is a great example of just using basic HTML to define your content.

Exploring the project structure: the home.ts file

This Typescript file contains very little code. In fact, this is the bare minimum to create an Ionic Page.

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

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

```
    constructor(public navCtrl: NavController) {

    }
}
```

We first import the Component module from the Angular core library. Just like the App component was standardized to Component, so was the Page component standardized also to Component. Next,we import the NavController component from the Ionic-Angular library. One of the biggest things you have to remember when you are developing with Angular2, is you have to import the components, and features from Angular and Ionic in order to use them. The entire library is no longer available for use by default.

Our Component decorator defines its templateUrl as 'home.html' and the selector to be page-home.

The Build Directory

Before Ionic RC0, the Ionic CLI when generating a running version of our app, built the finished app in a slightly different manner. It used to take all the HTML files and copy them into a build directory within the www directory. Any changes to Sass files -- the .scss files -- that we have generated will be compiled and appended to each platform's CSS file. Our TypeScript is transpiled to ES5 and packed with the Angular and Ionic code and saved as *app.bundle.js*.

Although there is still a build directory within the www, you will not find your HTML files within it anymore. Now, when the Ionic CLI generates a running version of our app, the HTML templates are pre-compiled and included within the main.js file. The build directory now stores four files: *main.css*, *main.js*, *main.js.map*, and *polyfills.js*. All of the .scss files are compiled and merged into main.css. Our TypeScript is still transpiled into ES5 and packed with the Angular and Ionic code and saved as *main.js*. The *main.js.map* files assists with any debugging we may need to do, while the *polyfill.js* address any cross-browser compatibility issues that need to be addressed.

The Theme Directory

The next top-level element within the src directory is the Theme directory. Here we find a Sass file that we can modify to override any default CSS used by Ionic. We won't spend much time here now, other than to say, with Ionic2 the ability to customize our app's visual styling is much easier.

The Assets Directory

This directory was added in RC0, the goal of this directory is to house the various assets your app can need like; fonts, images, and the like. Since Ionic can be used to

create Progressive Web Apps (PWAs), a default *manifest.json* file is included. We will discuss this file and PWAs in more detail in a later chapter.

The declarations.d.ts file

Because TypeScript utilizes static types, we need to be able to "describe" code we want to use and import. This is handled through type definitions, and there is a fairly large collection of them managed by the TypeScript team. If for some reason you are unable to find the types for the third-party library, you can create a short-hand type definition in the *declarations.d.ts* file. The *.d.ts* denotes that the file is a definition file and not actual code. Within the file, we can add a line to declare our module:

```
declare module 'theLibraryName';
```

This line does is tell the Typescript compiler that the module is found, and it is an object of any type. This will allow the library to be used freely without the Typescript compiler giving errors.

The service-worker.js file

The final item in the src directory is the *service-worker.js* file, another component of creating PWAs. We will ignore it for now, and return to it in earnest later.

Now that we have had a brief survey of the basic files created by the Ionic CLI for this template, let's get to work and modifying them for our Ionic2Do app.

Updating the Page Structure

While Home might be acceptable for some projects, I want to update our structure to be more reflective of what this page is, a list of our tasks. Let's go through our files and directories and update the reference to tasklist. We will start with the *app.component.ts* file.

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

import { TaskListPage } from '../pages/tasklist/tasklist';

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

  constructor(platform: Platform) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      StatusBar.styleDefault();
    });
```

```
  }
}
```

Visual Studio Code

If you are using Code as your editor, you will see portions of your code get a red squiggly line underneath it. This is editor informing you of an issue; like a missing file or a unreferenced variable.

We also need to update the app.module.ts file as well. Since we are adjusting the starting component, we need to adjust app.module.ts file to reflect the new component name and directory.

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { TaskListPage } from '../pages/tasklist/tasklist';

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

Save this file, and we will move on to adjusting the directory structure and file names. Within the page directory change the references from 'home' to 'tasklist'.

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

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

export class TaskListPage {

  constructor(public navCtrl: NavController) {
```

```
  }
}
```

If you are still running `$ ionic serve`, stop it by entering q in the terminal. Then run `$ ionic serve` again, and we should see no visible changes from before.

Let's first update our HTML template by opening *tasklist.html*.

In the `<ion-title>` tag, change Blank Starter to Tasks. Next, we add a button to the header that will allow us to add tasks to our list. Adding buttons in either a navbar or it's parent component, the toolbar, is a bit different from adding standard buttons.

After the closing `</ion-title>` tag, add a `<ion-buttons>` tag. This component acts as a container for a collection of buttons. This component did not exist in Ionic 1, causing some interesting solutions when you wanted a row of buttons within your navbar.

Within this tag we need to included an attribute that controls the placement of the buttons it contains. You can either give it the value of 'start' or 'end'. I want our add item button to be placed at the far left of the header, so set this to 'end'.

```
<ion-buttons end></ion-buttons>
```

Next, we can add a standard button element <button> and use the Angular2 syntax define the click handler to a function named addItem(). We will also add the ion-button directive to give the button the proper platform styling. Another change in Ionic 2, are the **icon-left** and **icon-right** directives. These directives will add a small amount of padding to either the left or right of the icon. Otherwise, the icon will appear directly next to the text. If you have an icon only button, then add **icon-only**. This will add padding to all the sides of the icon to give it proper spacing.

```
<ion-buttons end>
  <button ion-button icon-left (click)="addItem()">Add Item</button>
</ion-buttons>
```

Go ahead and save this file. If `$ ionic serve` is running, we should see our changes applied in the browser.

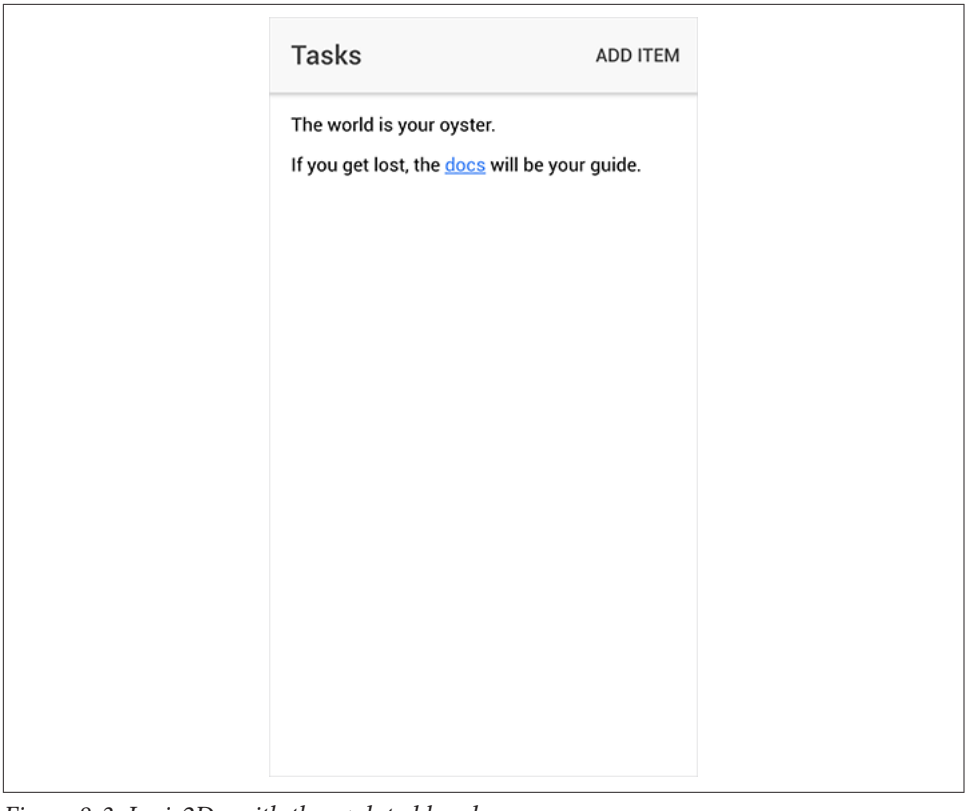


Figure 8-3. Ionic2Do with the updated header

Ionic ships with a nice collection of icons. They are actually hosted as an independent Github project, so you can use them outside of Ionic if you want to.

Ionic 2 made several changes on how we work with the Ionicon library. First, we now have the `<ion-icon>` tag, instead of using `<i>` . Next, instead of referencing the icon we want to use by defining it with a CSS class, we set just the name attribute to the icon we want. For the Add Item icon, a standard + icon will work just fine. The markup is:

```
<button ion-button icon-left (click)="addItem()">
  <ion-icon name="add"></ion-icon> Add Item
</button>
```

Many icons have both Material Design and iOS versions. Ionic will automatically use the correct version based on the platform.

However, if you want more control, you can explicitly set the icon to use for each platform. Use the `md` (material design) and `ios` attributes to specify a platform-specific icon:

```
<ion-icon ios="logo-apple" md="logo-android"></ion-icon>
```

If you want to use a specific icon, then simply use the icon's full name. For example, if we wanted to use the iOS outline map icon across all platforms the markup would be:

```
<ion-icon name="ios-map-outline"></ion-icon>
```

To find the full icon name, go to and http://ionicframework.com/docs/v2/ionicons/and select the icon. It will display the icon information.

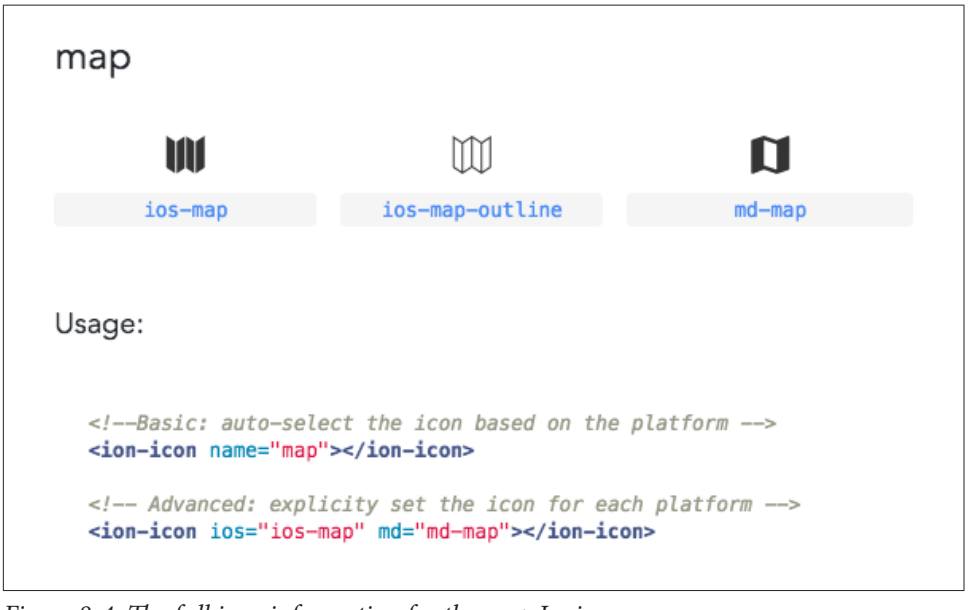


Figure 8-4. The full icon information for the map Ionicon.

For a complete list of the icons and their names, visit http://ionicframework.com/docs/v2/resources/ionicons/

Now, let's turn our attention to building our list of tasks.

Replace the placeholder content in the `<ion-content>` with this bit of HTML.

```
<ion-content>
  <ion-list>
    <ion-item *ngFor="let task of tasks">{{task.title}}</ion-item>
  </ion-list>
</ion-content>
```

Let's walk through this code fragment. The `<ion-content>` tag serves as our container for content. It will automatically scroll if the content exceeds the viewport. Next, we have the `<ion-list>`, which as you might guess is used to display rows of information. In our case, this will be rows of tasks. Within our <ion-list> we define what our list item's template will be, and how to map and bind our data.

With Angular 1, we would use the ng-repeat directive to define the array to repeat through to generate our tasks. Angular 2 changed this directive to *ngFor. It does essentially the same function. For our sample, we are going to loop through an array named tasks and set each item into a locally scoped variable named task. Using the data-binding syntax, {{task.title}}, we will render out each task's title string. Save this file, and open the *tasklist.ts* file.

Within the class, we will define the tasks variable. Since we are using TypeScript, we need to set the type. For now, let's just define the tasks array as an array of generic Objects.

```
export class TaskListPage {
  tasks: Array<Object> = [];

  constructor(public navCtrl: NavController) {
  }
```

Now, we need to include an actual constructor in our class definition.

Within the constructor, let's set our tasks array to some placeholder content to verify that our template code is working:

```
constructor(public navCtrl: NavController)  {
   this.tasks = [
      {title:'Milk', status: 'open'},
      {title:'Eggs', status: 'open'},
      {title:'Syrup', status: 'open'},
      {title:'Pancake Mix', status: 'open'}
   ];
}
```

Save this file. Again, using `$ ionic serve` , we can preview this running app in our browser.

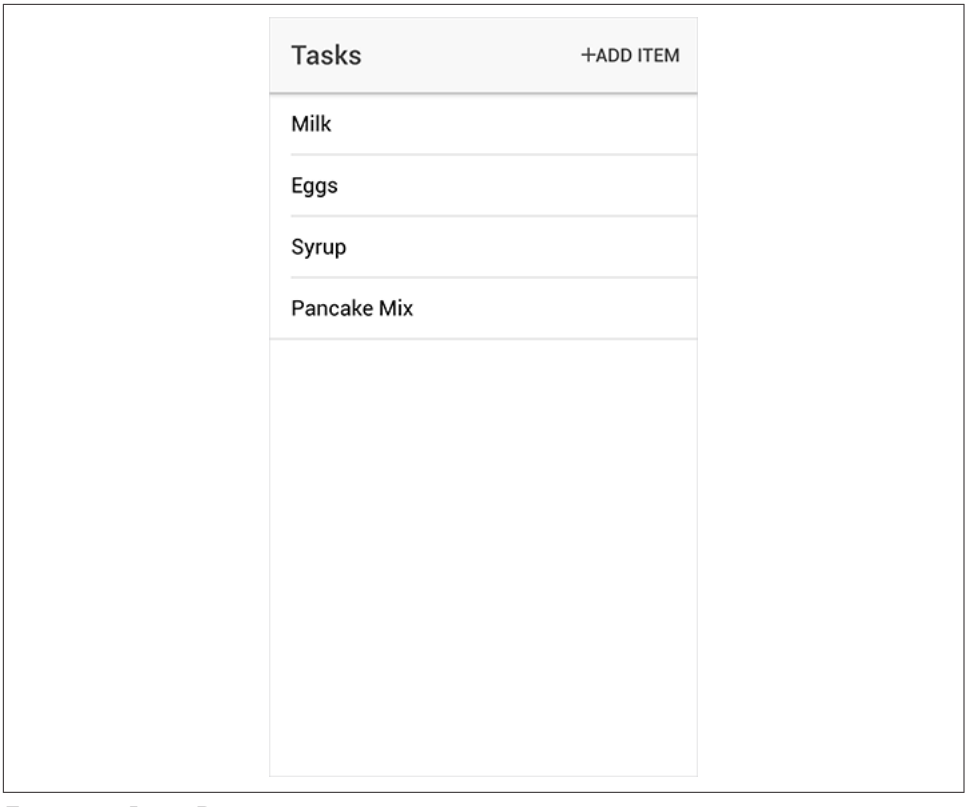


Figure 8-5. Ionic2Do app

Let's turn our attention to being able to add an item to our task list.

If you recall, we already included a click handler, addItems in the HTML file, so we now need to write it.

In the *tasklist.ts* file, after the end of the constructor we will add our addItem function. For now, we will use the standard prompt method to display a dialog to allow the user to enter a new task title. This will be included in a generic object that is pushed onto our tasks array.

```
addItem() {
  let theNewTask: string = prompt("New Task");
  if (theNewTask !== '') {
    this.tasks.push({ title: theNewTask, status: 'open' });
  }
}
```

Save the file and wait for the app to be recompiled and reloaded. Switching back to our browser, click on the Add Item button in the header and the browser will open a

prompt dialog, enter a new task and close the dialog. Angular will automatically update our list.

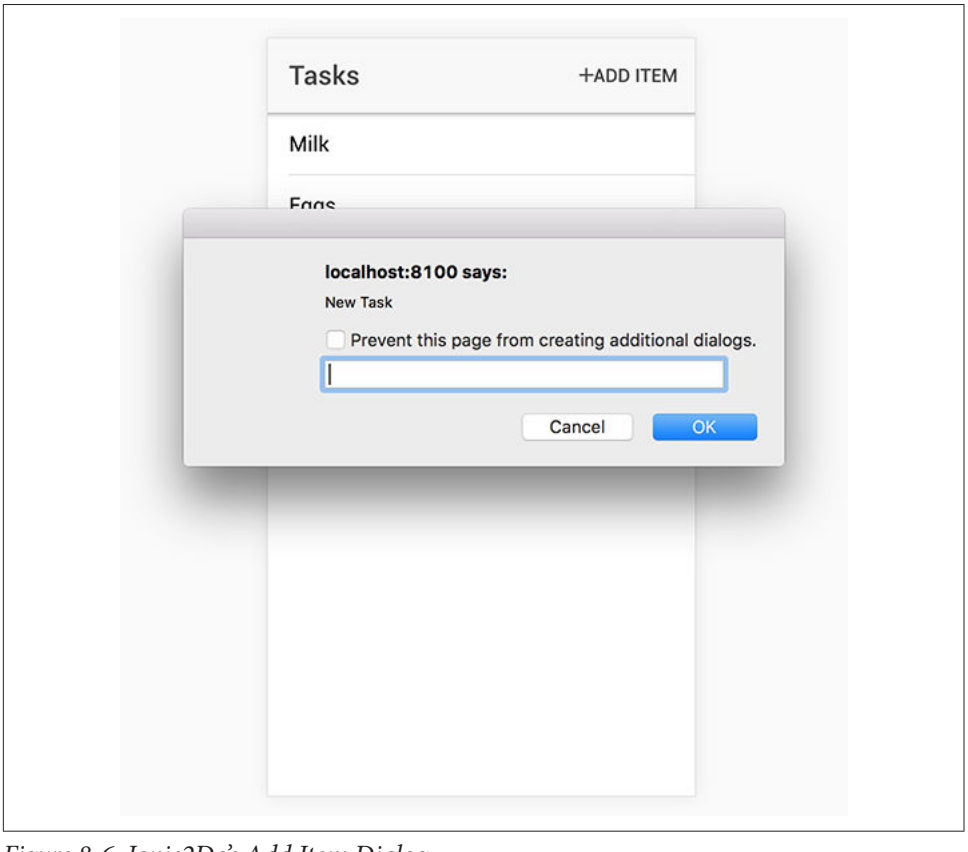


Figure 8-6. Ionic2Do's Add Item Dialog

If you launched your preview using `$ ionic serve --lab` , you noticed that only one list was updated with your new tasks. That is because **ionic serve** is actually running two instances of the app.

Now, that we can add items to our task list, let's add a method to mark them as done. A common interface pattern for this is to swipe from right to left on a row to reveal a set of buttons. Ionic 2 provides a `<ion-item-sliding>` component that recreates this user experience.

Replace this code block

```
<ion-item *ngFor="let task of tasks">
     {{task.title}}
</ion-item>
```

with

```
<ion-item-sliding #slidingItem *ngFor="let task of tasks">
  <ion-item>
    {{task.title}}
  </ion-item>
</ion-item-sliding>
```

The **ngFor** is now placed on the `<ion-item-sliding>` component instead of the `<ion-item>`. The elements within the `<ion-item>` tag will be our visible row items. We also need to include the reference to the slidingItems variable.

Next, we need to use the `<ion-item-options>` component to contain our buttons that will be shown when we swipe the row. This component supports having these option buttons be on the right, left or even both sides. Simply add **side='right'** to the ion-item-options that you want revealed with the user swipes from the right to left. For items that you want to show when the user swipes from left to right, define it as **side='left'**. If you do not include a side, it will default to the right side.

For this app, we will have a button to mark a task as done, and another button to remove it from the list completely. The markup is just the standard `<button>` tag. Each button will have a click function and use an icon from the IonIcon library. Here is the snippet:

```
<ion-list>
  <ion-item-sliding *ngFor="let task of tasks">
    <ion-item>
      {{task.title}}
    </ion-item>
    <ion-item-options side="right">
      <button ion-button icon-only (click)="markAsDone(task)" color="secondary">
        <ion-icon name="checkmark"></ion-icon>
      </button>
      <button ion-button icon-only (click)="removeTask(task)" color="danger">
        <ion-icon name="trash"></ion-icon>
      </button>
    </ion-item-options>
  </ion-item-sliding>
</ion-list>
```

To mark a task as done, our click handler will call a function named 'markAsDone', and pass it the reference to that row's task. If you have used Angular before, you know this is a great example of the power of the framework. You can let Angular handle keeping track of each row and let it resolve the management of what task we are interacting with, rather having to do all the bookkeeping ourselves.

For the button content, we will just use the `<ion-icon>` component. Now, since there is nothing inside the `<ion-icon>` , you might be tempted to self-close this tag. But Angular requires the tags within a template not be self-closed.

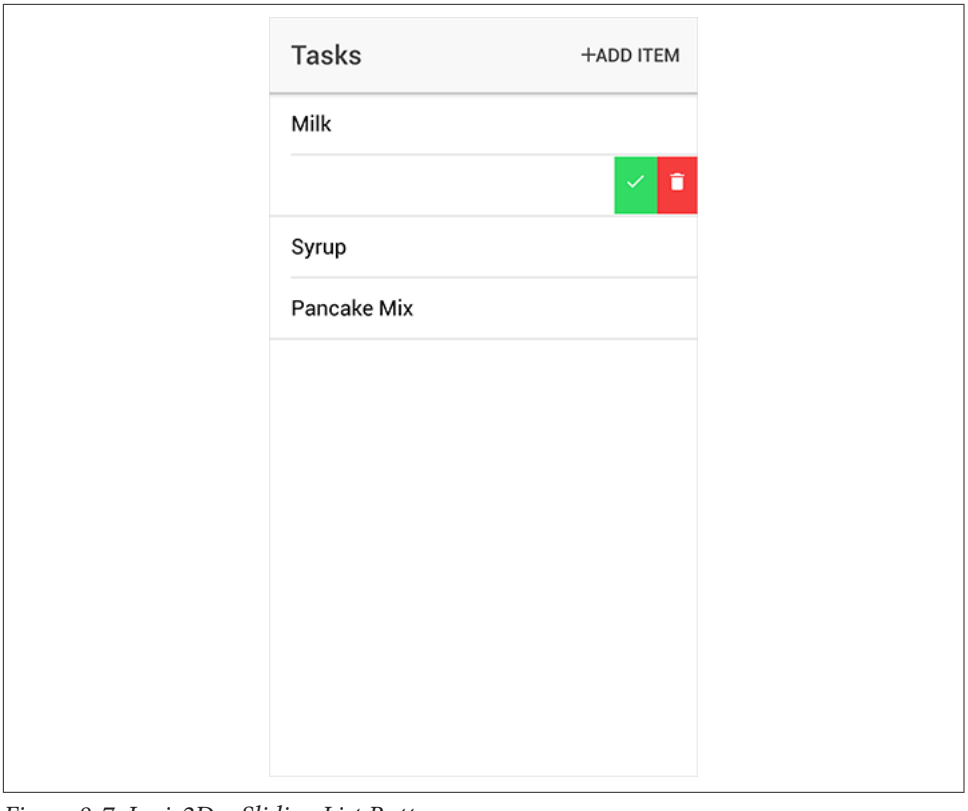


Figure 8-7. Ionic2Do: Sliding List Buttons

Let's switch to tasklist.ts and add our two functions; **markAsDone** and **removeTask**, after our addItem function.

```
markAsDone(task: Object) {
   task.status = "done";
}

removeTask(task: Object) {
   task.status = "removed";
   let index = this.tasks.indexOf(task);
   if (index > -1) {
      this.tasks.splice(index, 1);
   }
}
```

Before we test our app, let's make it so that when a user marks a task as done, we draw a line through the task. CSS makes this very easy with the text-decoration property. In the tasklist.scss file, add the following CSS:

```
.taskDone {text-decoration: line-through;}
```

Angular provides a method of conditionally applying CSS classes to an element. We will have Angular apply the CSS class, *taskdone* if the task's status property is done to our <ion-item>

```
<ion-item [ngClass]="{taskDone: task.status == 'done'}" >
```

Angular 1 to Angular 2

This directive is a good example of some of the subtle changes between Angular 1 and Angular 2. In Angular 1, the syntax was ng-class="expression". In Angular 2, the syntax is now [ngClass] ="expression"

Make sure all the files are saved, then test the app in your browser. You should be able to swipe a row and reveal the two option buttons. Click on the checkmark to mark the task as done. You should see the text change to have a line drawn through it. But, the row did not slide back over the option buttons after we clicked. Let's fix this interface issue.

In the *tasklist.html* file, we first, we need to modify our import statement to include the ItemSliding component.

```
import {NavController, ItemSliding} from 'ionic-angular';
```

Then we need to modify the `<ion-item-sliding>` component. Add a local template variable, *#slidingItem*. The <ion-item-sliding> component will set this variable with a reference to each row. Next, pass this variable as the first parameter to both our option button functions. The new code will look like this:

```
<ion-item-sliding *ngFor="let task of tasks" #slidingItem>
  <ion-item [ngClass]="{taskDone: task.status == 'done'}">
    {{task.title}}
  </ion-item>
  <ion-item-options side="right">
    <button ion-button icon-only (click)="markAsDone(slidingItem, task)" color="secondary">
      <ion-icon name="checkmark"></ion-icon>
    </button>
    <button ion-button icon-only (click)="removeTask(slidingItem,task)" color="danger">
      <ion-icon name="trash"></ion-icon>
    </button>
  </ion-item-options>
</ion-item-sliding>
```

Save the file, and open the *tasklist.ts* file again. We need to include our new parameter to both the markAsDone and removeTask functions. This parameter will be typed to List.

```
markAsDone(slidingItem: ItemSliding, task: Object) {
  task.status = "done";
  slidingItem.close();
```

```
}

removeTask(slidingItem: ItemSliding, task: Object) {
  task.status = "removed";
  let index = this.tasks.indexOf(task);
  if (index > -1) {
    this.tasks.splice(index, 1);
  }
  slidingItem.close();
}
```

We can call the close method on the list reference, and it will slide our list item back for us.

Ionic 1 to Ionic 2

Ionic 1 also had a similar list component. But to trigger the close action required injecting a reference to $ionicListDelegate and using it. Ionic 2 cleans up this code and just requires the reference to the list's row be available.

Try our new version out, and you will see that our rows will close after we click on either option button.

Adding Full Swipe Gesture

You probably have used the full-swipe gesture to perform an action on a list item. A great example of this is in the iOS Mail app. If you do a short swipe on the list, the option buttons will reveal themselves. This is what we have working in our application now. But in the iOS Mail app, if you keep swiping, the last option element is automatically triggered. Let's add this to our application.

First, we need to add an event listener for the ionSwipe event. This event will be triggered when the user has performed the full-swipe gesture. We also need to tell it what function to call. Typically, this would be the same function that the last option button would call. So, our `<ion-item-options>` will now become:

```
<ion-item-sliding *ngFor="let task of tasks" #slidingItem ↵
  (ionSwipe)="removeTask(slidingItem, task)">
```

Second, we need to add an additional property on the button that we want to visually expand as the gesture is performed. Again, since this is typically done on the last item, we will add to our button that delete's the task. The new button is now:

```
<button ion-button icon-only expandable (click)="removeTask(slidingItem, task)" danger>
   <ion-icon name="trash"></ion-icon>
</button>
```

And with those two additions, we have added support for a full-swipe gesture in our application.

Simple Theming

Besides adapting the components look to the platform, we can also quickly affect their color. Ionic has five pre-defined color themes: primary (blue), secondary (green), danger (red), light (light gray), and dark (dark gray). To apply a theme color, we can just add to most Ionic components. Let's add a touch of color to the header, by setting it's theme color to primary

```
<ion-navbar color="primary">
```

We had already set the theme color to each of the option buttons.

```
<button color="secondary" (click)="markAsDone(slidingItem, task)">
…
<button color="danger" (click)="removeTask(slidingItem, task)">
```

We will explore styling our Ionic Apps in further detail in a later chapter. But for now, we have a little color in our app.

Proper Typing

If you were carefully watching the output in your console, you should have noticed a few warnings. It should have said something like:

```
src/pages/tasklist/tasklist.ts(28,10): error TS2339:
Property 'status' does not exist on type 'Object'.
```

This is the TypeScript informing us about the fact we add a property on to the Task object. Now this error did not prevent our app from working, but we should address it.

The fix is quite simple, we need to create a custom Task class, and use it instead of the generic Object. We could just define the Task Class directly within the tasklist.ts file, but instead we will create a new file named *task.ts*. In this file, we will export our class named Task. It will have two properties; title and status, both typed as Strings.

```
export class Task {
  title: string;
  status: string;
}
```

In the tasklist.ts file, we need to inject this class definition.

```
import { Component } from '@angular/core';
import { NavController, ItemSliding } from 'ionic-angular';
import { Task } from './task';
```

Then change the typing of our task array from

```
tasks: Array<Object> = [];
```

to

```
tasks: Array<Task> = [];
```

We also need to adjust the input parameters for both the markAsDone and removeTask functions. The task variables need to be typed as Task.

Upon saving, the TypeScript compiler should no longer be generating warnings in your console.

Saving Data

You might have noticed a major flaw in our app. Any tasks that we add are not being saved. If you reload the app, only the four initial tasks are shown. There are a lot of options we can use to solve this problem. We could use localStorage to save our data, but there are cases where this data can be cleared out by either the user or the system. WebSQL is an option, but the specification is no longer maintained, so there might be long term support issues to consider. If we want to use a Cordova plugins, there are several other options to consider. Since our data structure is fairly straightforward it could be written to the file system as a simple text file, and read back in. Not very elegant solution (nor secure), but it would work. A more advanced option might be to use the SQLite Plugin, maybe in conjunction with PouchDB. We could also look at using the new Storage module from Ionic itself.

But these are one to one solutions. In our connected, multi-device, multi-platform world, I want to be to share and interact with my task list across all my mobile devices.

Creating a FireBase account

One option you can use to quickly have a cloud-based database is FireBase from Google. One of Firebase's services is to provide a real-time JSON database for storing and sync your app's data. If you do not have an account, you can sign up for a free developer account at https://firebase.google.com/.

There you will be prompted to create a new project, go ahead give your project the name of Ionic2Do. Next, select your country or region from the list and click Create Project. This will create a basic Firebase system for us to use. From this dashboard screen, click the Database choice, which will display our default database.

We will need several configuration elements to connect our application up to Firebase. Firebase makes this fairly simple, locate the Add Firebase to your web app button and click it. This will display a complete code sample for connecting our app, however, we are only interested in the config values:

```
var config = {
  apiKey: "your-api-key",
  authDomain: "your-authdomain",
  databaseURL: "https://someurl.firebaseio.com",
  storageBucket: "someurl.appspot.com",
  messagingSenderId: "your-sender-id"
};
```

Save this information to a temporary file, as we will need this information later.

But before we can use this database, we need to adjust it's security setting. Click the Rules tab, to bring up the Rules editor. By default, the database is set to require authorization for both read and write access. Since we are just working on a simple tutorial, we can relax these settings. Changes the rules from:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

to

```
{
  "rules": {
    ".read": "auth == null",
    ".write": "auth == null"
  }
}
```

Then click Publish. Our database can now be read and written to without needing authentication.

Installing Firebase and AngularFire2

In order to use Firebase, we need to install some addition node modules that our app will use.

Everything is in Beta

Both the AngularFire and Firebase libraries are still under active development. The code samples listed here might change as the libraries are updated. Please refer to either the O'Reilly site or ionic2book.com for updates.

The first node module we need to install is the @types module. In the early development of TypeScript, there were several different type declaration managers for the .d.ts files. With the release of TypeScript 2.0, the preferred method is to use the @types module to manage any additional library declarations.

```
$ npm install @types/request@0.0.30 --save-dev --save-exact
```

Next, we need to update the *tsconfig.json* file to support proper typing for any Firebase methods we will be using.

Within the compileOptions, we need to add a typeRoots parameter and have it point to our newly install @types location. We also need to include a types parameter and have it include firebase as a type. Here is the complete *tsconfig.json*:

```
{
  "compilerOptions": {
    "allowSyntheticDefaultImports": true,
    "declaration": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [
      "dom",
      "es2015"
    ],
    "module": "es2015",
    "moduleResolution": "node",
    "target": "es5",
    "typeRoots": [
      "../node_modules/@types"
    ],
    "types": [
      "firebase"
    ]
  },
  "exclude": [
    "node_modules"
  ],
  "compileOnSave": false,
  "atom": {
    "rewriteTsconfig": false
  }
}
```

Now, let's install Firebase into our project. From the command line:

```
$ npm install firebase  --save
```

Although we could work with Firebase directly, the AngularFire library was built to make this process even easier. Now with the addition of Observables in Angular2, the method to interface with Firebase has become even simpler. If you have worked with Angular 1 and Firebase, then you probably used the AngularFire library to provide AngularJS bindings for FireBase. AngularFire 2 is the TypeScript based version of this library. This version is still in early beta at the time of this writing. To install the AngularFire library, execute this from the command line:

```
$ npm install angularfire2 --save
```

Ionic Build Systems

When we use the Ionic CLI to build our application, it executes a series of scripts collectively known as the Ionic App Scripts. The Ionic App Scripts were recently broken out from the main framework as a stand alone GitHub repo, so they can be updated and improved independently of the framework itself. These scripts are what takes our various files, third party libraries, assests and whatever else our apps needs and produces our runable application. Throughout the development of Ionic 2, the team

changed the build system four times; from Browserify to WebPack to Rollup.js and back to WebPack. As of this writting, Ionic 2 is supporting both Rollup.js and Webpack as build options. The exact method for doing this is undergoing changes. We recommend following the Ionic Blog for the latest updates on the Ionic build options.

The impact of these different build systems is how you integrate third party modules into your code base. Since our Ionic2Do app needs to integrate Firebase and AngularFire 2, the process is different for each build system.

Rollup.js Adjustments

In order to use Firebase and AngularFire2 with a Rollup.js build system, we need to make some changes to the *rollup.config.js* file. This file can be found in node_modules/@ionic/app-scripts/config/ directory. These changes will tell Rollup.js to properly include our libraries.

There are two changes that need to be made to this script. The first is to change the global useStrict setting from true to false. This is due to the use of eval within the Firebase library. Hopefully, the need for this change will be resolved and we now longer need to include it in our script. The second change to expand the parameters that are being passed into commonjs. We need to include the RxJS, FireBase and AngularFire2 libraries, as well as the namedExports. Here is the complete rollup.config.js file:

```
var nodeResolve = require('rollup-plugin-node-resolve');
var commonjs = require('rollup-plugin-commonjs');
var globals = require('rollup-plugin-node-globals');
var builtins = require('rollup-plugin-node-builtins');
var json = require('rollup-plugin-json');


// https://github.com/rollup/rollup/wiki/JavaScript-API

var rollupConfig = {
  /**
   * entry: The bundle's starting point. This file will
   * be included, along with the minimum necessary code
   * from its dependencies
   */
  entry: 'src/app/main.dev.ts',

  /**
   * sourceMap: If true, a separate sourcemap file will
   * be created.
   */
  sourceMap: true,

  /**
   * format: The format of the generated bundle
   */
```

```
  format: 'iife',

  /**
   * dest: the output filename for the bundle in the buildDir
   */
  dest: 'main.js',

  // Add this to avoid Eval errors in Firebase
  useStrict: false,

  /**
   * plugins: Array of plugin objects, or a single plugin object.
   * See https://github.com/rollup/rollup/wiki/Plugins for more info.
   */
  plugins: [
    builtins(),
    commonjs({
        include: [
        'node_modules/rxjs/**',
        'node_modules/firebase/**',
        'node_modules/angularfire2/**'
        ],
        namedExports: {
        'node_modules/firebase/firebase.js': ['initializeApp', 'auth', 'database'],
        'node_modules/angularfire2/node_modules/firebase/firebase-browser.js': ↵
          ['initializeApp', 'auth', 'database']
        }
    }),
    nodeResolve({
      module: true,
      jsnext: true,
      main: true,
      browser: true,
      extensions: ['.js']
    }),
    globals(),
    json()
  ]

};


if (process.env.IONIC_ENV == 'prod') {
  // production mode
  rollupConfig.entry = '{{TMP}}/app/main.prod.ts';
  rollupConfig.sourceMap = false;
}

module.exports = rollupConfig;
```

Warning

Be aware if you update the Ionic App Scripts, your changes may be lost. Always keep a safe backup of this file.

Webpack Adjustments

If you are building your application with Webpack, there are no special changes needed to integrate AngularFire2/Firebase into the application. We do however need to install a different app script. To do this, run the following command:

```
$ npm install @ionic/app-scripts@beta
```

Another note about building with Webpack, the TypeScript compiler that it uses tends to be a bit stricter than the one used by Rollup.js system. As such, your build might fail due to TypeScript errors. So we highly advise you to keep the terminal window visible while you compile to spot this issues.

Updating our app.module.ts file

Regardless of which build system, we are using, there are some changes we need to make to our app.module.ts file to use our AngularFire/Firebase solution. First, we need to import the actual components from the angularfire2 package.

```
import { AngularFireModule } from 'angularfire2';
```

The second change is to define our default Firebase configurations. Before the @NgModule declaration, add this code: (replacing the values that were assigned to you when you created your Firebase account)

```
export const firebaseConfig = {
  apiKey: "your-api-key",
  authDomain: "your-authdomain",
  databaseURL: "https://someurl.firebaseio.com",
  storageBucket: "someurl.appspot.com",
  messagingSenderId: "your-messageSenderID"
};
```

The final change is to the imports array within the @NgModule declaration. Here we need to add the call to the AngularFireModule and initialize it with our config.

```
imports: [
  IonicModule.forRoot(MyApp),
  AngularFireModule.initializeApp(firebaseConfig)
],
```

With that, our initial setup to use AngularFire/Firebase is done. Let's turn our attention to where the real work lies in the *tasklist.ts* file.

Updating the tasklist.ts file

Like with the changes to *app.module.ts*, the first thing we need to update is our import statements. We need to import the AngularFire and FirebaseListObservable modules from the AngularFire 2 library.

```
import { AngularFire, FirebaseListObservable } from 'angularfire2';
```

Typically when working with dynamic data, a common solution is to use the Observable module from the RXJS library. If you have not heard of the RXJS library before, it is a set of libraries to compose asynchronous and event-based programs using observable collections and Array#extras style composition in JavaScript. The library is actively maintained by Microsoft. You can learn more about this library at https://github.com/Reactive-Extensions/RxJS.

AngularFire takes the use of Observable a bit further and extends to a custom version, FirebaseListObservable. This is what we will use for our application.

Next, we need to replace using a local array to store our tasks, to one that will be bound to the Firebase data.

So our `tasks: Array<Task> = [];` will now become `tasks: FirebaseListObservable<any[]>;` . This will enable any updates to be applied to our tasks variable.

Our component is going to be using the AngularFire library to communicate with our Firebase database. In order to do this, we will need to pass a reference to that library as a parameter in our constructor.

```
constructor(public navCtrl: NavController, public af: AngularFire) { … }
```

Since our list of tasks is going to be stored remotely, we can replace our initial tasks array with the AngularFire request to return the data in the tasks sub-directory.

```
this.tasks = af.database.list('/tasks');
```

At the moment, this array should be empty since our database is empty. Before we can go ahead and perform a spot-check on our app by running `$ ionic serve` , we need to add a Pipe to our ngFor directive in the *tasklist.html*. In order to use the FirebaseListObservable array, we need to tell Angular that this data will be asynchronously fetched. To do so, we just need to include the async pipe. Otherwise, when Ionic/Angular tries to render this template, there isn't any data defined yet and it will throw an error. By adding the async pipe, Angular knows how to properly handle this information delay.

```
<ion-item-sliding *ngFor="let task of tasks | async" #slidingItem
  (ionSwipe)="removeTask(slidingItem, task)">
```

What is a Pipe?

Pipes are functions that will transform data within a template. Angular has several built-in pipes to perform common tasks like changing text case or formatting numbers.

You should be able to add new items to your task list. After you have added a few items, go ahead and quit the server, then relaunch it. You should see your items repopulate your list.

Alternatively, your could log into your Firebase account, and see your data listed.

You might have noticed that we did not modify the addItem function for this to work. Since our tasks variable is a FirebaseListObservable to our Firebase database, adding new items is still done via the push method.

However, both the **markAsDone** and **removeTask** methods will need some refactoring in order to properly interact with our Firebase. Instead of directly interacting with the specific array item, we must use the AngularFire methods to do this.

In the **markAsDone** function, replace

```
task.status = "done";
```

with

```
this.tasks.update(task.$key, { status: 'done' });
```

The **task.$key** is the unique key value that is generated by Firebase when it was added to the database.

In the **removeTask** function, replace

```
task.status = "removed";
let index = this.tasks.indexOf(task);
if (index > -1) {
  this.tasks.splice(index, 1);
}
```

with

```
this.tasks.remove(task.$key);
```

Save our file, and run our application again using `$ ionic serve`. Our sliding buttons will now properly update our Firebase dataset.

You might have noticed that the task.$key references are being flagged with the following warning: Property '$key' does not exist on type 'Task' any. This is a simple fix. Open the *task.ts* file and add a new property $key and set its type to any.

With that, we have transformed our local To Do application to one that used a cloud-based one with very few lines of code. We have just barely touched on the power of

Firebase and AngularFire2. It would be worth your time exploring the capabilities of these libraries further.

Ionic Native

Let's see how our app looks in an emulator. From the command line, either use `$ ionic emulate ios` or `$ ionic emulate android`.

Emulating Android

In case you forgot, launching the default Android emulator can be extremely slow. I recommend using Genymotion as a substitute for any virtual device testing .

Emulating iOS

When you run just ionic emulate ios, you might want to target a specific device type and OS. To do this, simply append the command with `--target="devicename, OSType"` . For example, if I wanted to target an iPhone 5s running iOS 10 the command would be: `ionic emulate ios --target="iPhone-5s, 10.0"`

Our app should connect with our FireBase database, and display our task list. Now, let's add a new task, by tapping the Add Item button.

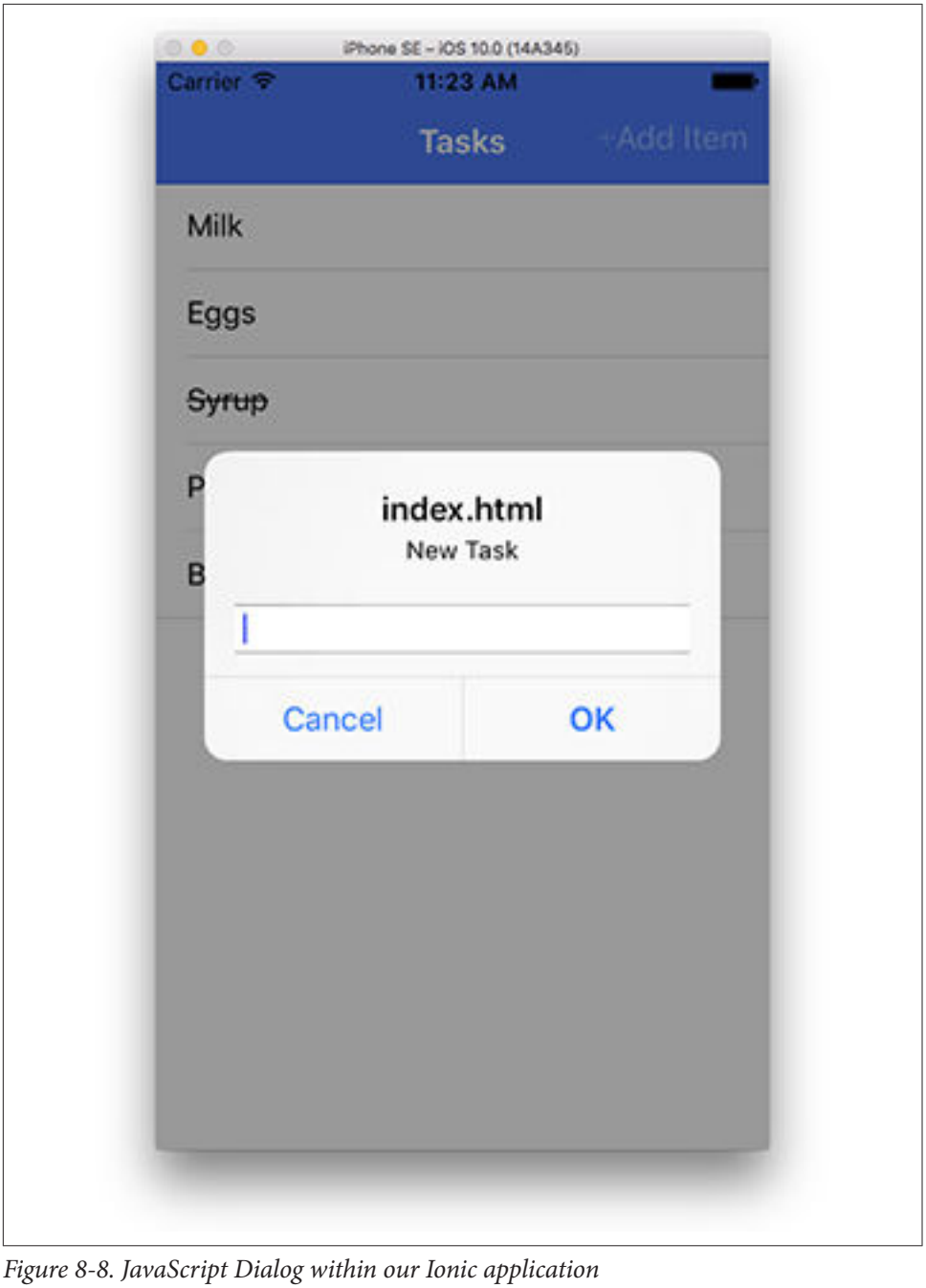


Figure 8-8. JavaScript Dialog within our Ionic application

Notice the dialog, it is the same JavaScript dialog that we have been seeing in our browser. Now if we want a native looking app, we certainly need to address this before releasing it to the app stores.

One of the many plugins that exist for Cordova is one to use native dialogs in place of the Javascript version. To support using Cordova plugins with the Ionic 2 framework, they create a curated set of wrappers for many of the most common plugins.

By default, Ionic Native is included, but if you need to manually include Ionic Native into our project, use this command:

```
$ npm install ionic-native --save
```

With Ionic Native installed we will have the interfaces needed to interact with our Cordova plugins.

What about ngCordova?

The ngCordova library (also maintained by the team behind Ionic), was designed for use with Angular 1 and Ionic 1 based projects. Ionic Native is the replacement solution for Angular 2 and Ionic 2 based projects.

The next step we need to perform is to install the actual plugin. One of the available plugins for Cordova is the Dialogs plugin. This plugin will render native dialogs in our app, instead of the web based options. From the terminal we can install it via npm:

```
$ ionic plugin add cordova-plugin-dialogs
```

This will download the plugin code for all our installed platforms and update the config.xml so our app will now be built with this code reference included.

Plugin Sources

In the past, the plugins were referenced via the org.apache.cordova.* naming system. When the plugin host was moved to npm, the naming system changed to the cordova-plug-* pattern. If you might find references to the older system in some documentation, so you will want to replace that reference with the proper npm version of the plugin.

In the *tasklist.ts*, we will need to import the Dialogs module. With the other import statements, add the following

```
import {Dialogs} from 'ionic-native';
```

Then we need to replace these lines of code

```
let theNewTask: string = prompt("New Task");
this.tasks.push({ title: theNewTask, status: 'open' });
```

with

```
Dialogs.prompt('Add a task', 'Ionic2Do', ['Ok', 'Cancel'], '')
.then(
  theResult => {
    if (theResult.buttonIndex == 1) {
      this.tasks.push({ title: theResult.input1, status: 'open' });
    }
  }
)
```

Index Values

You might be wondering why we are comparing to an index value of 1 since typically arrays start counting from 0. The zero index used by any dismissal of the dialog not by using the buttons themselves.

If we want to error-proof our code a bit, we could check that the inputted string was not empty before we added it our list. Here is what the application's Add Item Dialog now looks like

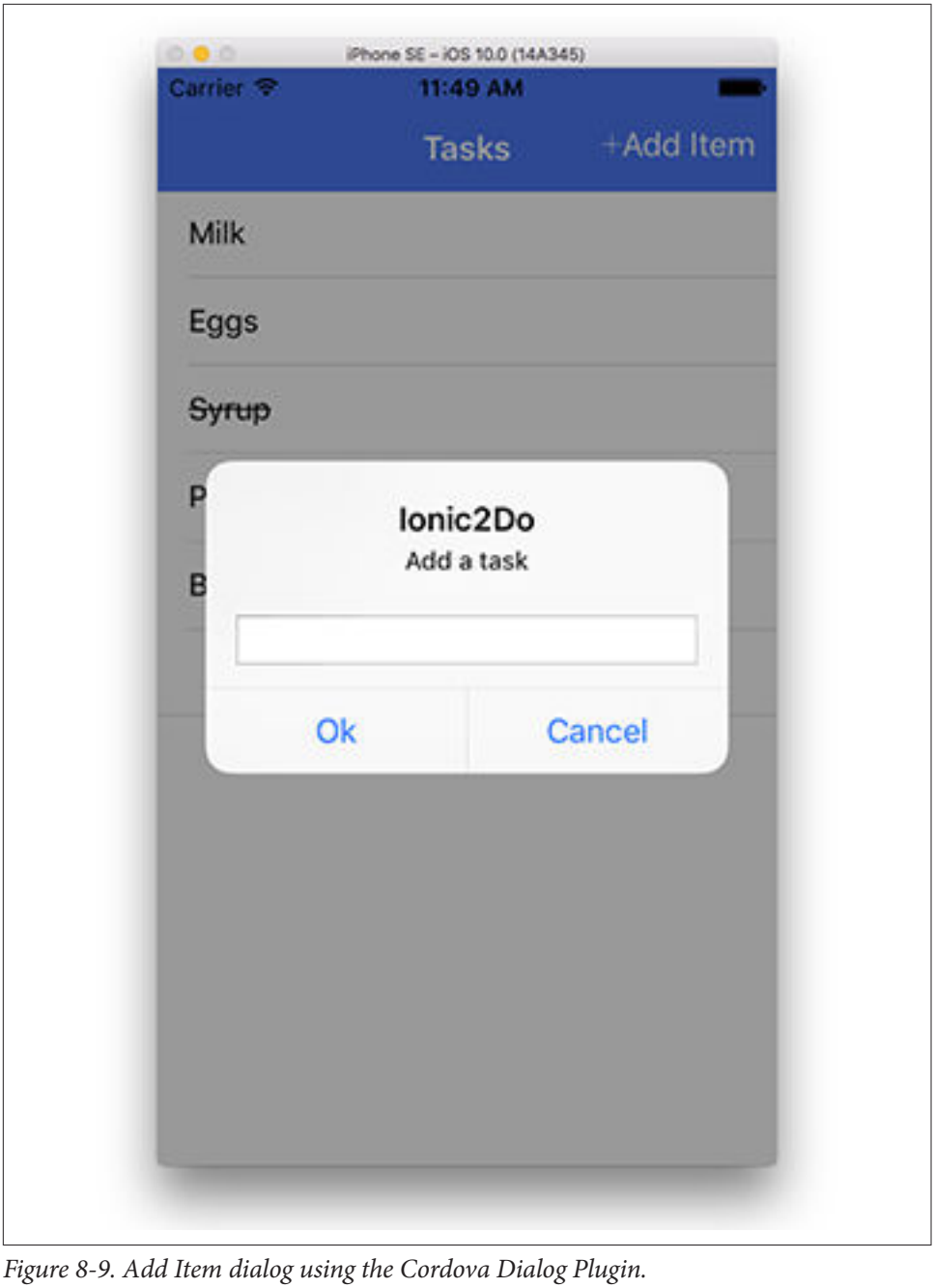


Figure 8-9. Add Item dialog using the Cordova Dialog Plugin.

And with that was a basic To Do application built. We looked at some basic Ionic components and theming, working with an external library, and incorporated the

Ionic Native library. In the next chapter, we will explore more Ionic components and navigation methods.

CHAPTER 9

CHAPTER 10

Building a Tab-based App

A very common type of application you might build is one that uses a tab-based navigation system. This design pattern works very well when you have a limited number (5 or fewer) of groups (or tabs) of content. We are going to use design pattern to create an app that will allow you to explore the various U.S. National Parks (in honor of them celebrating their centennial in 2016). Here is a what our final app will look like.

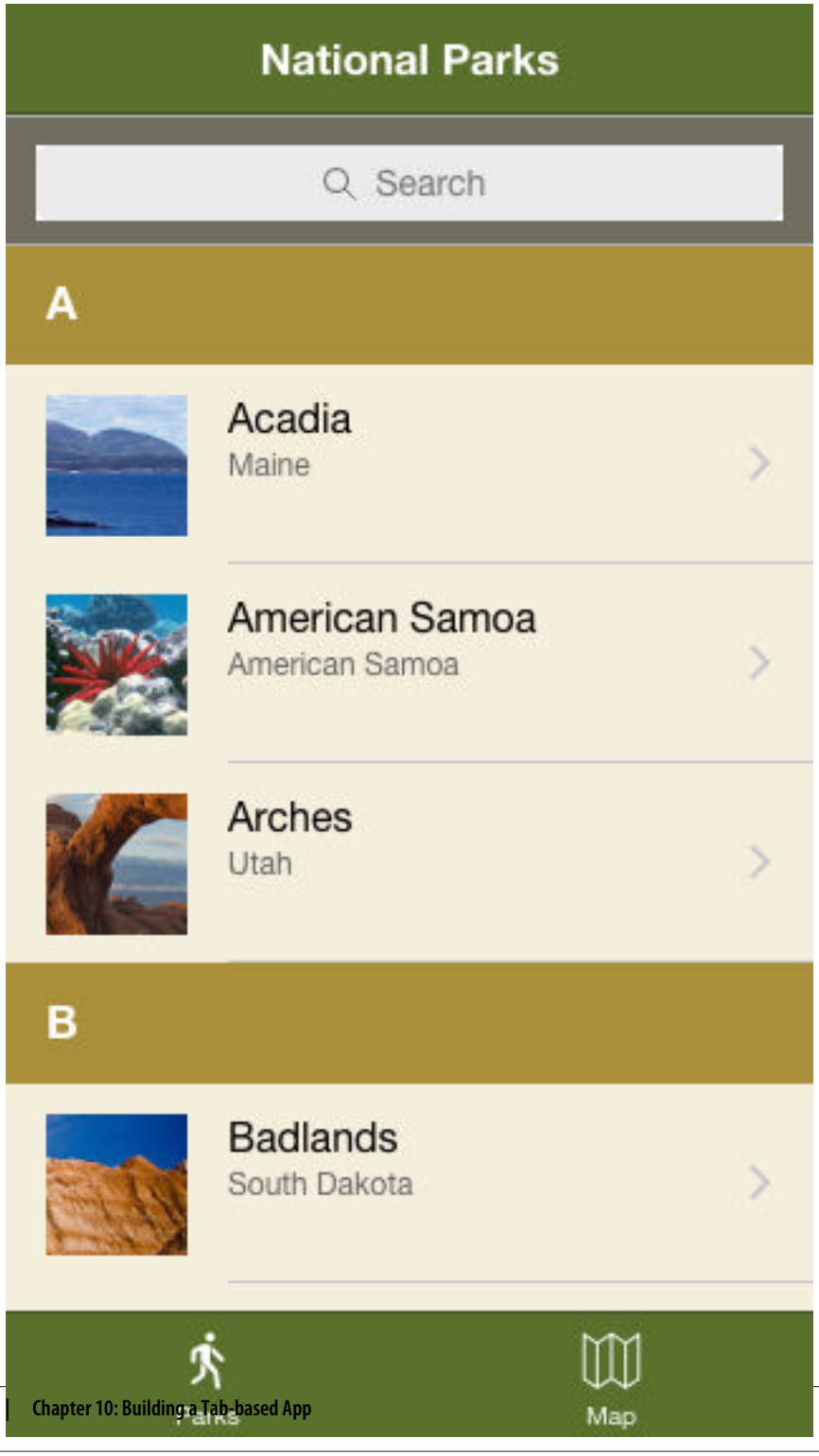


Figure 10-1. Ionic National Parks App

We are again going to use the Ionic CLI to scaffold our application. First, create a new directory that we will be building from. I named my directory IonicParks.

```
$ ionic start IonicParks --v2
```

Since we are going to be creating a tabs based app, we did not need to pass in a template name, since the tab template is the default.

Next, change your working directory to the newly created IonicParks directory

```
$ cd IonicParks
```

Now let's explore the template itself before we get down to business.

```
$ ionic serve
```

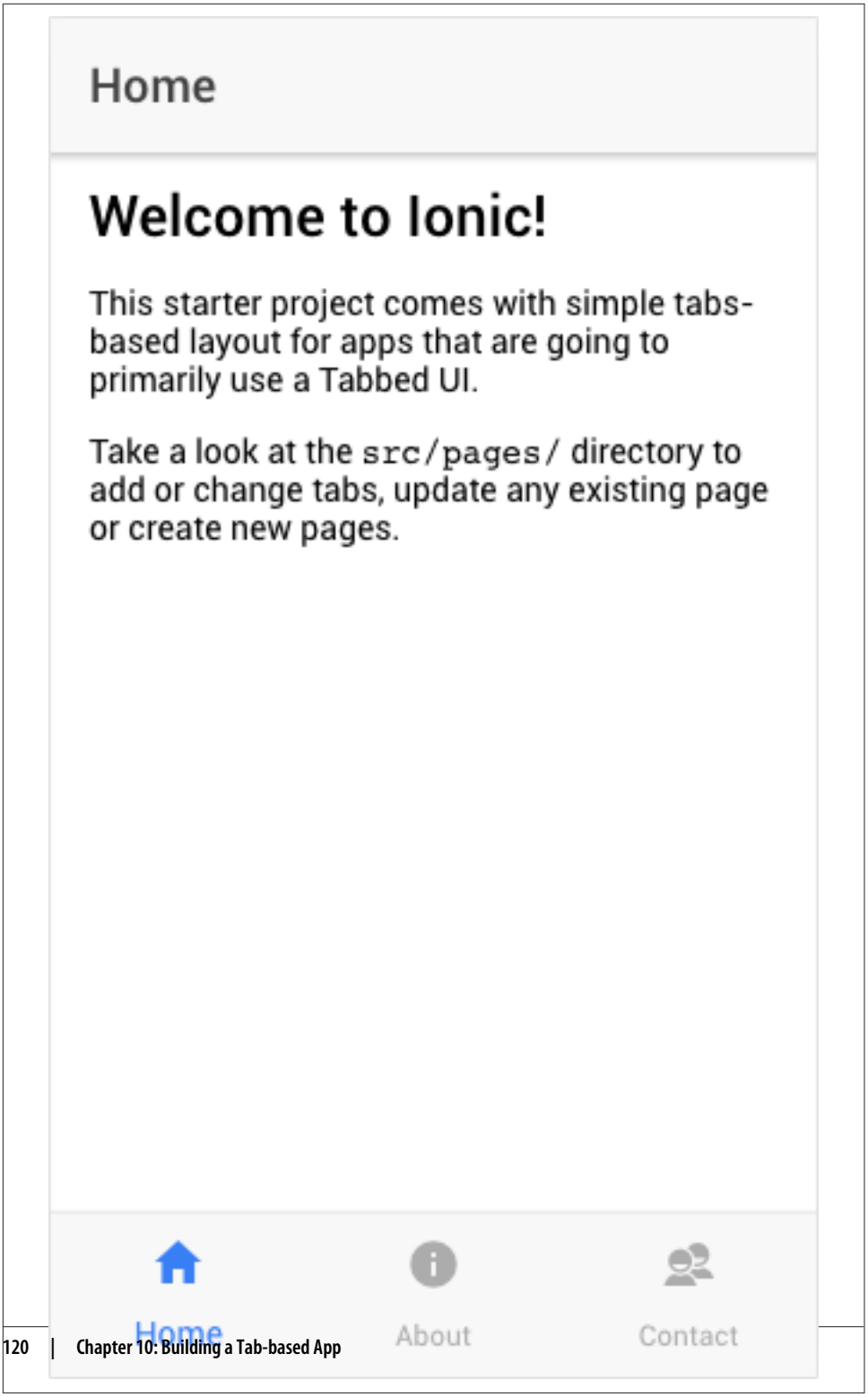


Figure 10-2. Ionic Tabs Template.

Not a lot here, but we can navigate between the three tabs (named Home, About and Contact), and see the content change.

Taking a look inside the *app.module.ts* file and we see that instead of import one page, we now are importing four. These four pages comprise the three pages for each tab (HomePage, AboutPage and ContactPage) and one page (TabsPage) that will serve as the container for the application. Each of these pages is included in the declaration and entryComponents array.

Looking at the *app.component.ts* file, the only change here is that the rootPage is the TabsPage and not a specific tab view.

Now, let's go take a look at the pages directory. Inside this directory, we will find four additional directories: about, contact, home, and tabs. Open the home directory, and within it are the HTML, SCSS and TS files that define our home tab.

The SCSS file is just a placeholder, with no actual content inside. The HTML file has a bit more content inside.

First, the HTML defines an `<ion-navbar>` and `<ion-title>` component:

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

Next, the code defines the `<ion-content>` tag. The component also sets the padding directive within the `<ion-content>` tag.

```
<ion-content padding>
```

The rest of the HTML is just plain vanilla HTML.

Now, let's take a look at the component's TypeScript file. This file is about as light-weight as possible for an Ionic Page.

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

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

  constructor(public navCtrl: NavController) {

  }
}
```

First, we import the Component module into our code. Next, our NavController is imported from the ionic-angular library. This component is used whenever we need to navigate to another screen.

In our Component decorator, we set our templateURL to the HTML file and the selector to page-home.

The last bit of code is just setting the HomePage class to be exported and has the nav-Controller pass into the constructor.

The other two pages are almost identical to this Page component. Each exporting out a component respectively named AboutPage and ContactPage.

Let's look at the tabs themselves. In the tabs directory, we see that it contains just a *tabs.html* file and a *tabs.ts* file. Let's look at the HTML file first.

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home">↵
  </ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="About" tabIcon="information-circle">↵
  </ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts">↵
  </ion-tab>
</ion-tabs>
```

Like Ionic 1, Ionic 2 also has an `<ion-tabs>` and `<ion-tab>` components. The <ion-tabs> component, is just a wrapper component for each of its children, the actual tabs themselves. The <ion-tab> component has more few attributes that we need to change. Let's start with the two easier ones; **tabTitle** and **tabIcon**. These two attributes set the text label of the tab and the icon that will be displayed. The icon names are from the IonIcon library.

You do not need to set both the title and the icon on tab component. So depending on how you want yours to look, only include what you want.

Also, depending on the platform you are running your app on, the tab style and position will automatically adapt.

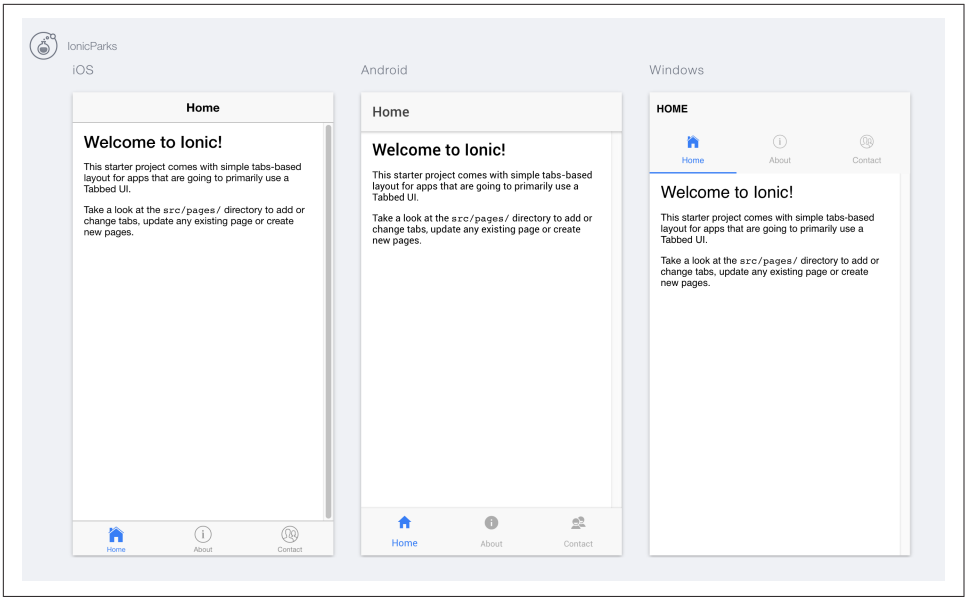


Figure 10-3. Tabs component rendering based on platform type.

If you want to force a specific tab placement, there are two options, either directly on the component itself with:

```
<ion-tabs tabsPlacement="top">
```

Or globally through the app config options. This is done in the *app.module.ts* file.

```
IonicModule.forRoot(MyApp, {tabsPlacement: 'top'} )
```

For a full list of configuration options, visit http://ionicframework.com/docs/v2/api/config/Config/.

The last item to look at in the <ion-tab> is the [root] binding that defines what Component should act as that tab's root.

```
import { Component } from '@angular/core';
import { HomePage } from '../home/home';
import { AboutPage } from '../about/about';
import { ContactPage } from '../contact/contact';

@Component({
  templateUrl: 'tabs.html'
})
export class TabsPage {
  // this tells the tabs component which Pages
  // should be each tab's root Page
  tab1Root: any = HomePage;
  tab2Root: any = AboutPage;
  tab3Root: any = ContactPage;
```

```
    constructor() {

    }
}
```

We have our now familiar import statements. The first loads the base Component module, the next three load each of the pages for our tabs.

Next, in our Component decorator, we set the templateUrl to the *tabs.html* file.

The class definition is where we assign each of the tabs to their corresponding components. That is all we need to have to establish our tabs framework. Ionic will manage the navigation state for each tab for us.

Bootstrapping our App

Now that we have a general understanding of the structure of a tab-based Ionic 2 application, we can start to modify it for our app. But, rather than having you go through all the files and folders and rename them to something meaningful, we are going to take a shortcut. I have already made the initial changes to the template files. In addition, I also included a data file with the National Park data and various images for use in the app.

First, go ahead and delete the IonicPark directory that the CLI generated. Now, we will be scaffolding our app from a template from my GitHub repo.

```
$ ionic start Ionic2Weather https://github.com/chrisgriffith/Ionic2Weather --v2
```

Once this process is complete, again remember to change your working directory:

```
$ cd IonicPark
```

and if you targeting Android, don't forget to add that platform:

```
$ ionic platform add android
```

Loading Data via the HTTP Service

Before we start building the app in earnest, let's create a provider to load our local JSON data into our app. This way, we will have actual data to work with as we build out our app.

In the app directory, create a new directory named providers, and within that directory create a new file named *park-data.ts*.

As you can expect, using the HTTP service in Angular 2 is slightly different from Angular1. The main difference is that Angular 2's HTTP service returns observables through RxJS, whereas $http in Angular 1 returns promises.

Observables give us expanded flexibility when it comes to handling the responses coming from the HTTP requests. As an example, we could leverage a RxJS operator

like retry, so that failed HTTP requests are automatically re-sent. This can be very useful if our app has poor, weak or intermittent network coverage.

Since our data is being loaded locally, we don't need to worry about that issue.

Returning to our *park-data.ts* file, we will inject three directives into it.

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

Next, we use the @Injectable() decorator. The Angular 2 documentation reminds us to remember to include the () after the @Injectable, otherwise, your app will fail to compile.

We are going to define ParkData as the provider's class. Now add a variable named data, and define its type to any and set it to be null. In the constructor, we will pass in the Http directive that we imported, and classify it as a public variable. The classes' actual constructor is currently empty.

```
@Injectable()
export class ParkData {
  data: any = null;

  constructor(public http: Http) {}

}
```

Within the class definition, we will create a new method named load. This method will do the actual loading of our JSON data. In fact, we will add in some checks to make sure we only load this data once throughout the lifespan of our app. Here is the complete method:

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

  return new Promise(resolve => {
    this.http.get('assets/data/data.json')
      .map(res => res.json())
      .subscribe(data => {
        this.data = data;
        resolve(this.data);
      });
  });
}
```

Since we are hard coding the file that we are loading, our method is not taking in a source location, hence load().

The first thing this method does is check if the JSON data had been loaded and saved into the data variable. If it has, then we return a Promise that resolves to this saved data. We have to use a promise since that is the same return type that the actual loading portion uses. Let's look at that block of code.

If the data has not been loaded, we return a basic Promise object. We set it to resolve after call the HTTP service and using the get method from our JSON data that is at *assets/data/data.json*. As with all promises, we need to define its subscription. Again using the fat arrow function, =>, the result is stored in a variable named res. We will take the string and use Angular's built-in JSON converter to parse it into an actual object. Finally, we will resolve our promise with the actual data.

At this point, we only have written the provider. Now, we need to actually use it in our app. Open *app.component.ts* , and add `import { ParkData } from '../providers/park-data';` with the rest of the import statements. This will load the provider and assign it to ParkData.

Next, we need to add an array of providers in the @Component declaration.

```
@Component({
  template: `<ion-nav [root]="rootPage"></ion-nav>`,
  providers: [ ParkData ]
})
```

Let's now modify the constructor to load our data. We will pass in the reference to the ParkData provider into the constructor. After the platform.ready code block, we will call the **parkData.load()** method. This will trigger the provider to load our data file.

```
constructor(platform: Platform, public parkData: ParkData) {
  platform.ready().then(() => {
    // Okay, so the platform is ready and our plugins are available.
    // Here you can do any higher level native things you might need.
    StatusBar.styleDefault();
  });

  parkData.load();
}
```

If you want to do a quick test to see if the data is being loaded, wrap the parkData.load() in a console.log(). The **__zone_symbol__value** of the **ZoneAwarePromise** will contain an array the 59 objects that are created.

What is a Zone?

A zone is a mechanism for intercepting and keeping track of asynchronous work. Since our data is being loaded in an asynchronous fashion, we need to use zones to keep track of the original context that we made our request in.

Here is a sample of what the each of the park's object data looks like:

```
createDate: "October 1, 1890"
data: "Yosemite has towering cliffs, waterfalls, and sequoias in a diverse↵
      area of geology and hydrology. Half Dome and El Capitan rise from the↵
      central glacier-formed Yosemite Valley, as does Yosemite Falls, North↵
      America's tallest waterfall. Three Giant Sequoia groves and vast ↵
      wilderness are home to diverse wildlife."
distance: 0
id: 57
image: "yosemite.jpg"
lat: 37.83
long: -119.5
name: "Yosemite"
state: "California"
```

Display our Data

Now that the data has been read into the app, and available via our service provider, let's turn our attention to actually displaying our 59 National Parks in a list.

First, we need to add a method on our Park Data service provider to actually return the data that we have loaded. After the load method, add the following method.

```
getParks() {
  return this.load().then(data => {
    return data;
  });
}
```

Two things to note about this method. First, it calls the load method. This is a safety check to ensure that the data is there. If for some reason it is not, it will load it for us. Since our provider is using promises, constructing a system that can be chained together is quite easy. Second, since this system is Promise based, we have to handle everything in a **.then** syntax. This is something that you might have to remember to do as you migrate from Angular 1 to Angular 2.

Switching to *park-list.html*, we the following after the <ion-content> tag:

```
<ion-list>
  <ion-item *ngFor="let park of parks" ↵
            (click)="goParkDetails(park)" detail-push>
    <ion-thumbnail item-left>
      <img src="assets/img/thumbs/{{park.image}}">
    </ion-thumbnail>
    <h2>{{park.name}}</h2>
    <p>{{park.state}}</p>
  </ion-item>
</ion-list>
```

If you recall from the Ionic2Do app, we will define an **<ion-list>** component, then define the **<ion-item>** that will be auto-generated for us. The repeated items are

being supplied by an array named parks, which we will define shortly. Each element of this array is put into a local variable named park. A click handler is also added to the **<ion-item>**, it will call a function named **goParkDetails** and will pass in the **park** variable as its parameter.

If our app is running in iOS mode, disclosure arrows will automatically be added. While on Android and Windows this icon is not added to the list item. If we want to show the right arrow icon that does not display it by default, we can include the **detail-push** attribute. Conversely, if we don't want to show the right arrow icon, we can use the **detail-none** attribute.

Within the `<ion-item>`, we will insert an `<ion-thumbnail>` component and set its position in the row by using item-left. The image tag is fairly straight forward. If you used the template for the project, it should have included an img directory that also contained a thumbs directory. That directory will hold thumbnails for each of our parks. By using Angular's data binding, we can dynamically set the src for each thumbnail with `src="assets/img/thumbs/{{park.image}}"` . Next, the park name and state are shown with a <h2> and <p> tags respectively and are also data bound to the park object.

One last thing to do is to remove the padding attribute on the `<ion-content>` as well. This will enable the list to be the full width of the viewport. With the HTML template updated, we can now focus on the component's code.

Extending parklist.ts

The first thing that we need to do is to inject our service provider into the component with

```
import { ParkData } from '../../providers/park-data';
```

Initially, the component's class is completely empty. We will replace it with the following code.

```
export class ParkListPage {
  parks: Array<Object> = []

  constructor(public navCtrl: NavController, public parkData: ParkData) {
    parkData.getParks().then(theResult => {
      this.parks = theResult;
    })
  }

  goParkDetails(theParkData) {
    console.log(theParkData);
  }
}
```

Let's define the parks variable that we referenced in our HTML template;

```
parks: Array<Object> = [];
```

Within the parameters of the constructor for the class, we will define a private variable parkData of type ParkData. As a general rule, it is always best to declare any variables to be private by default, and only when you need to expose them, should you change the access modifier to the variable.

Next, we will call the getParks method on the parkData. In the past, we might have written something like this to get our park data.

```
parks = parkData.getParks();
```

But since we are leveraging the power of Promises, we need to actually write our request for this data as such.

```
parkData.getParks().then(theResult => {
   this.parks = theResult;
   }
)
```

That wraps up the changes to the constructor itself. The last bit of code that was added was a placeholder function for the click event from the **<ion-item>**. The method accepts the park data object as a parameter, and simply writes that data to the console. We will be focused on this function shortly, but let's view our work so far by using `$ ionic serve`.

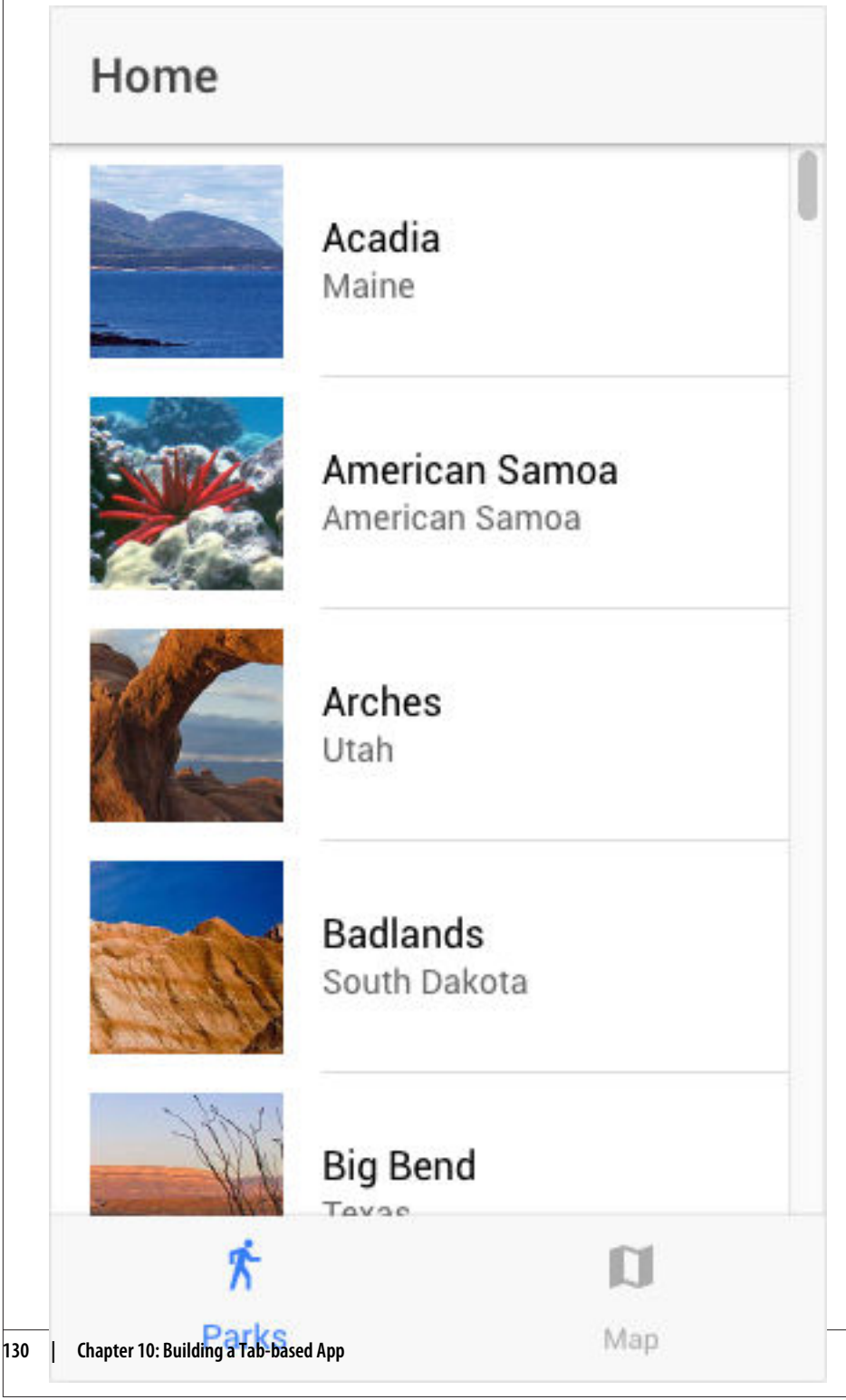


Figure 10-4. National Parks App

Once the app has been regenerated, we should see a scrolling list of National Parks in a browser, each with a thumbnail, title and state listed. If click on an item, the park data will be written out to the JavaScript console. Now that we have this initial screen working, we can turn our attention to creating the details page for that park.

Generating New Pages

With Ionic 2, the entire build process now more complex and structured, adding new pages into our app during development can be a bit tedious. Thankfully, there is a page generator function available in the CLI. Since we need to generate a park details page, our command will be

```
$ ionic g page parkDetails
```

The CLI will take our camelCase name and convert it into a kebab-case version. Let's go ahead and open it and make one small change. Since we have been appending all the page names with 'Page', we need to do so here as well.

```
export class ParkDetailsPage { ...
```

Another change we need to make is to include a reference to the component in the *app.module.ts* file.

```
import { ParkDetailsPage } from '../pages/park-details/park-details';
```

Then add this module to both the declarations and entryComponents arrays.

Ionic Generators

We also use the Ionic CLI generator to create providers as well, by replacing the page flag with the provider flag. In fact, the provider we wrote earlier in the chapter

Now, let's build upon the code in the *park-list.ts* file to enable the navigation to our newly generated page. We need to import some additional modules from the Ionic core. Our first import will become

```
import {NavController, NavParams} from 'ionic-angular';
```

Next, we will need to import the reference to the page that we are going to navigate to:

```
import {ParkDetailsPage} from '../park-details/park-details';
```

With these module injected into our component, the `goParkDetails` function can now become functioning.

```
goParkDetails(theParkData) {
  this.navCtrl.push(ParkDetailsPage, { parkData: theParkData });
}
```

Understanding the Ionic 2 Navigation model

Back in Ionic 1, the AngularJS UI-Router was used to navigate between pages. For many apps, this navigation system works fairly well. But if your application had a complex navigation model, using it would become problematic. For example, if we were creating our Ionic Parks app in Ionic 1, we would have to have two distinct URLs for a Parks Details page if we want to have accessible via both a park list screen and a map list screen. These types of issues forced the Ionic team to rebuild their entire navigation model.

The current navigation model is based on a push/pop system. Each new view (or page) is pushed onto a navigation stack, with each view stacking atop the previous one. When the user navigates back through the stack, the current view is removed (popped) from the stack.

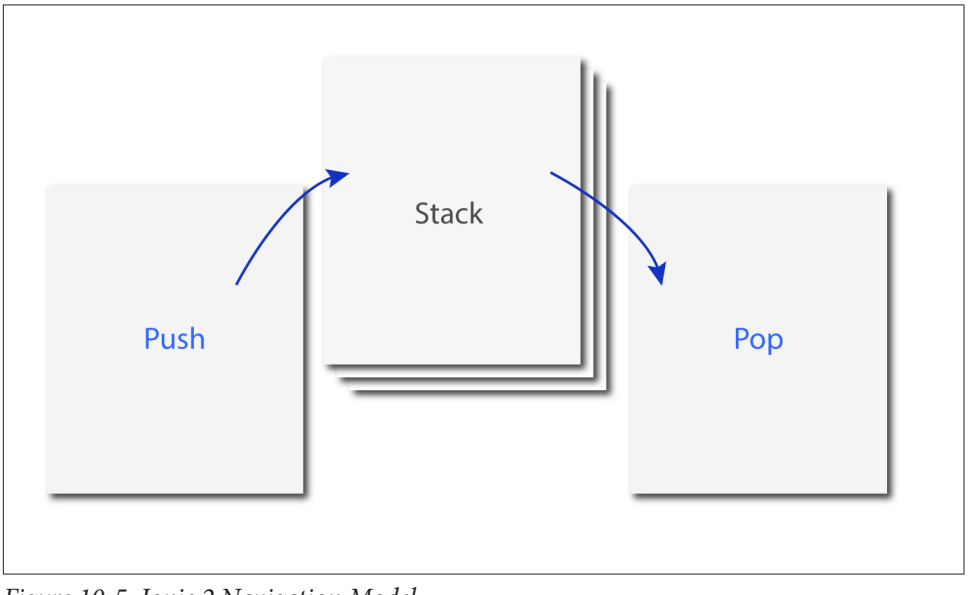


Figure 10-5. Ionic 2 Navigation Model

This new model makes the navigation model much simpler to work with.

Passing data between pages

In our **goParkDetails** function, it received the parkData for the clicked list item. By using the NavParams module, we can pass this data to the constructor of the new page.

We need to refactor the *park-details.ts* file to support the incoming data. First, we will need to import the NavParams module from ionic-angular. Next, in the class definition, we need to add a parkInfo variable that is typed to Object.

Now, we need to add a constructor to our class. It is in this constructor that the navigation parameters are passed in and stored in the variable navParams.

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

@Component({
  selector: 'page-park-details',
  templateUrl: 'park-details.html'
})
export class ParkDetailsPage {
  parkInfo: Object;
  constructor(public navCtrl: NavController, public navParams: NavParams) {
    this.parkInfo = navParams.data.parkData;
    console.log(this.parkInfo);
  }
}
```

For now, let's just write to the console the parkData that has been piggybacked on this parameter. Our selected park's data object is saved on the data method of the navParams. Saving our files, and running `$ ionic serve`, clicking on any item should now change our view and write to the console our data.

You will notice that the Ionic Framework, handled the screen to screen animations, as well as, automatically adding a back button in our header to enable the user to navigate back through the navigation stack.

Updating the Park Details Page

Since we can now navigate to the park details page, let's turn our attention to taking this dynamic data and displaying it. Here is what our completed Park Details will look like:

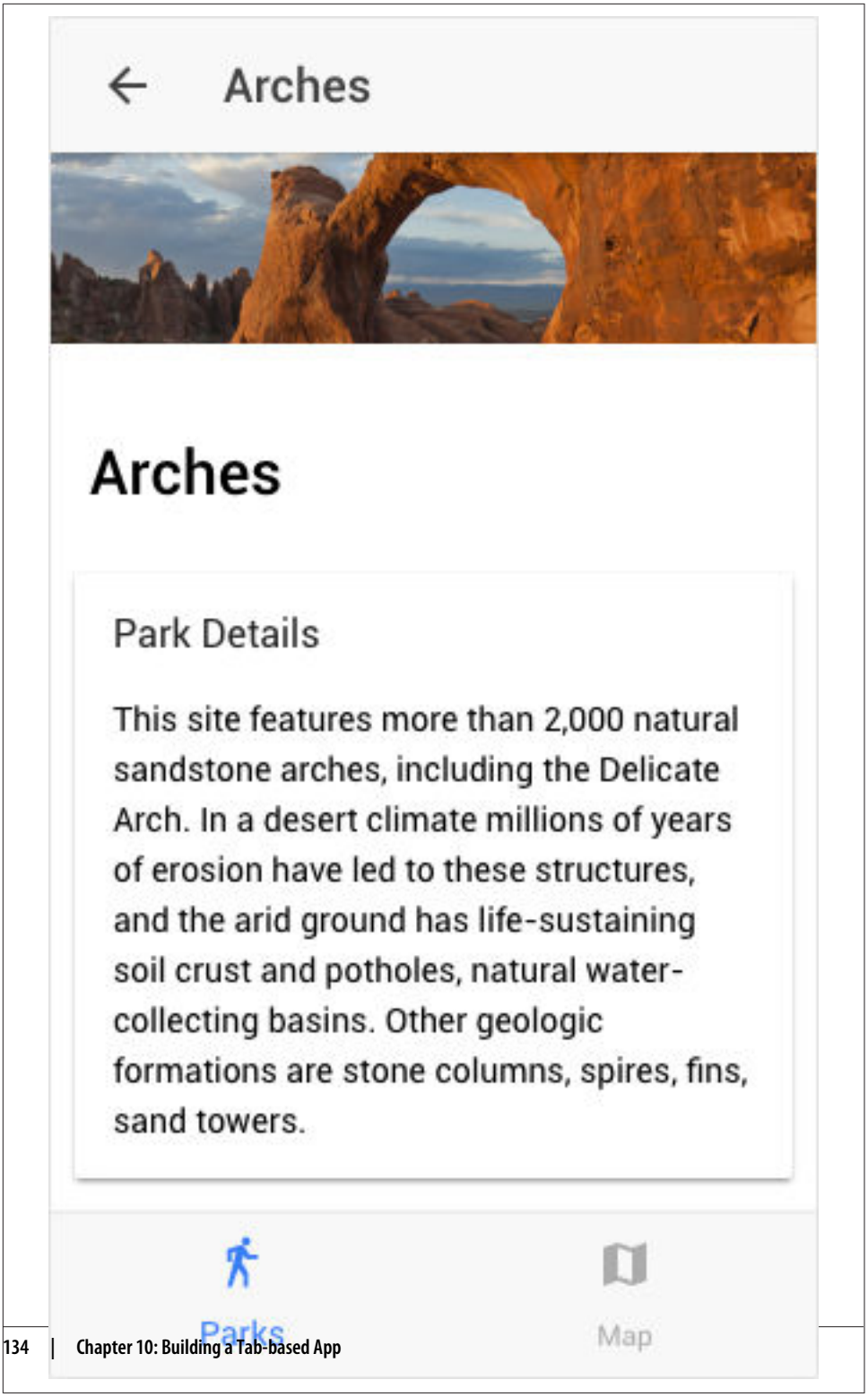


Figure 10-6. The National Park details screen.

The generated HTML page has some basics tags included, but we going to replace most it. First, let's remove the help comment from the top of the page. For the page title, we will replace it with {{parkInfo.name}}.

```
<ion-content class="park-details-page">
    <img src="assets/img/headers/{{parkInfo.image}}">
    <h1 padding>{{parkInfo.name}}</h1>
    <ion-card>
        <ion-card-header>
            Park Details
        </ion-card-header>
        <ion-card-content>
            {{parkInfo.data}}
        </ion-card-content>
    </ion-card>
</ion-content>
```

One new component we are using on this screen is the **<ion-card>**. As the documentation states, "Cards are a great way to display important pieces of content, and are quickly emerging as a core design pattern for apps". Ionic's card component is a very flexible container, supporting headers, footers, and a wide range of other components within the card content itself.

With a basic park details screen in place, go ahead and preview it with `$ ionic serve`.

Add a Google Map

As you might expect, an app about the national parks would require them each to be shown on a map of some kind. Unfortunately, there is not an official Google Maps Angular 2 module. There are some third party efforts, but let's work instead with the library directly. To do this we will need to include the library in our index.html file. Since the terms of use for the Google Maps SDK, forbids the storing of the tiles in an offline fashion, we can reference the remotely hosted version:

```
<!-- Google Maps -->
<script src="http://maps.google.com/maps/api/js"></script>

<!-- cordova.js required for cordova apps -->
<script src="cordova.js"></script>

<!-- The polyfills js is generated during the build process -->
<script src="build/polyfills.js"></script>

<!-- The bundle js is generated during the build process -->
<script src="build/main.js"></script>
```

We can ignore the need for an API key while we develop our application, but API key is required for production. You can obtain an API key at https://develop-

ers.google.com/maps/signup. When you get your API key change the script src to include it in the query string.

Adding Additional Typings

Since we are adding in a third party code library to be used in our app, wouldn't it be nice to have code hinting support and strong typing for that library? We can do this by extending our TypeScript definitions. The command to do this is:

```
$ npm install @types/google-maps --save-dev --save-exact
```

Adding our Content Security Policy

A Content Security Policy (CSP) is an added layer of security designed to reduce certain types of malicious attacks, including Cross-Site Scripting (XSS). Remember, our hybrid apps are still bound by the same rules that web apps have. As such, we also need to safeguard our applications in a similar manner.

In our *index.html* file, we need to include a CSP.

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

Since Google Maps transfers it's map tiles via the data uri method, our CSP needs to allow for this type of communication. In addition, we will need to add support of font-src as well as for the Ionicons to work. This should be placed within the <head> tag.

Adjust the CSS to support the Google Map

With our library able to be loaded and related data, let's turn our attention to map page itself. In *park-map.html*, we need to add a container for the map to be render in.

```
<ion-content class="park-map">
  <div id="map_canvas"></div>
</ion-content>
```

We need to give it either a CSS id or class, in order to apply some CSS styling. Since the tiles are dynamically loaded, our div has no width or height when it is first rendered. Even as the map tiles are loaded, the width and height of the container are not updated. To solve this, we need to define this div's width and height. In the *park-map.scss* file, add the following

```
#map_canvas {
    width: 100%;
    height: 100%;
}
```

This will give the container an initial value, and our map will be viewable.

Rendering the Google Map

We are going to work on this code in three sections. The first will be to get the Google Map displaying, the second will be to add markers for each of the National Parks, and the final section will make clicking on the marker navigate to the Park Details page. Switch to the *park-map.ts* file.

We will need to add two additional modules to the import statement from ionic-angular.

```
import {Platform, NavController} from 'ionic-angular';
```

We will use the Platform module, to make sure everything is ready before setting up the Google Map.

Within the class definition, we will define the map variable as a generic Object. This variable will hold our reference to the Google map.

```
export class ParkMapPage {
    map: google.maps.Map;
```

Next, we expand the constructor:

```
constructor(
  public nav: NavController,
  private platform: Platform,
  ) {
  this.map = null;
  this.platform.ready().then(() => {
    this.initializeMap();
  });
}
```

We make sure that we have reference to Platform module, then set up a Promise on the platform ready method. Once the platform ready event has fired, we then call our initializeMap function, using the fat arrow syntax.

```
initializeMap() {
  let minZoomLevel = 3;

  this.map = new google.maps.Map(document.getElementById('map_canvas'), {
    zoom: minZoomLevel,
    center: new google.maps.LatLng(39.833, -98.583),
    mapTypeControl: false,
    streetViewControl: false,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  });
}
```

This function will create the new map and assign it to the div with the id of "map_canvas". We also define some of the various map parameters. These parameters include; the zoom level, the center map (in our case, the center of the continental

United States), the various map controls, and finally the style of the map. The last object method is a custom method, where we will store the park information that we will need later in this chapter.

If we run `$ ionic serve` , then we should see a map being rendered in the Map tab.

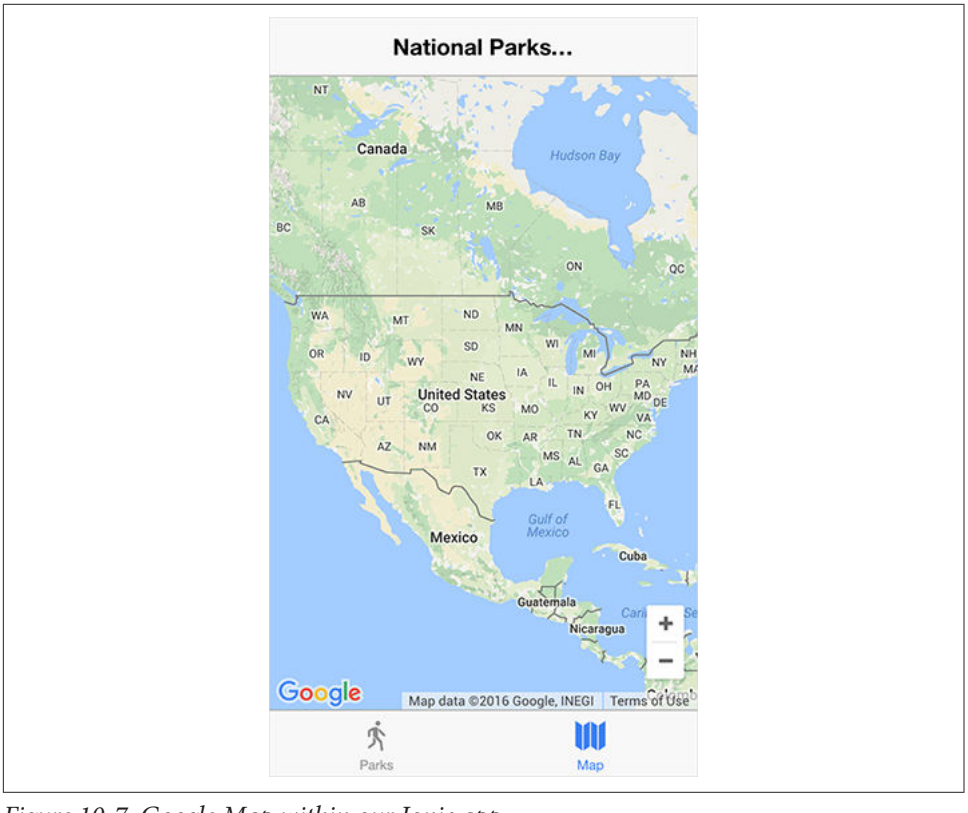


Figure 10-7. Google Map within our Ionic app

Add Map Markers

Now, that we have a Google map displaying in our mobile app, we can turn to the next task; add the markers for each National Park. The first thing we need to do is inject the ParkData service into our component.

```
import {ParkData} from '../../providers/park-data/';
```

Next, we will need to add an array that will hold our park data, as well as making sure the parkData is properly available to the class:

```
export class ParkMapPage {
  parks: Array<Park> = [];
  map: Object;
```

```
  constructor(
    public nav: NavController,
    public platform: Platform,
    public parkData: ParkData
    ) {
```

Although we could simply type our parks array to any, let's properly type to our park's data structure. To do this we will need to define the interface. Create a new directory named *interfaces* within the app directory. Within that new directory, create a new file named *park.ts*. This file will hold our simple definition for our Park interface. The code for this is:

```
export interface Park {
    id: number;
    name: string;
    createDate: string;
    lat: number;
    long: number;
    distance: number;
    image: string;
    state: string;
    data: string;
}
```

This interface will tell the compiler that Park data type will have these elements and their associated data types.

Back in the *park-map-page.ts* file, we will need to import this interface file.

```
import {Park} from '../../interfaces/park';
```

That should resolve any warnings in your editor about the Park data type.

Within the initializeMap function, we will need to add the code to actually display our markers.

But rather than use the standard Google marker image, let's use a marker that looks like the National Parks Service arrowhead logo.

```
let image = 'assets/img/nps_arrowhead.png';
```

Then we will get the park data from the parkData service. Once this promise is answered, the result will be stored in the parks array.

```
this.parkData.getParks().then(theResult => {
  this.parks = theResult;

  for (let thePark of this.parks) {
    let parkPos:google.maps.LatLng = ↵
        new google.maps.LatLng (thePark.lat, thePark.long);
    let parkMarker:google.maps.Marker = new google.maps.Marker();
    parkMarker.setPosition(parkPos);
    parkMarker.setMap( this.map);
```

```
      parkMarker.setIcon(image);
    }
  })
```

Our code will loop through this array and generate our markers. Save this file, and if ionic serve is still running, the app will reload. Select the Map Tab, and you now see icons on our map for each of the National Parks. Right now, these markers are not interactive. Let's add that capability.

Making the Markers Clickable

When a user clicks or taps on a marker on the map, we want to navigate them to the Park Details page for that markers. To do this we need to inject some of the Navigation modules from Ionic, as well as the actual ParkDetailsPage module. Our new import block will now look like this:

```
import {Component} from '@angular/core';
import {Platform, NavController, NavParams} from 'ionic-angular';
import {Park} from '../../interfaces/park';
import {ParkData} from '../../providers/park-data/park-data';
import {ParkDetailsPage} from '../park-details-page/park-details-page'
```

Within the for loop that adds each marker, we will need to add an event listener that will respond to our click, and then navigate to the ParkDetailsPage and pass along the marker's park data. Unfortunately, the standard Google Map Marker has none of that information. To solve this, we are going to create a custom Map Marker that we can store our park information.

Create a new file, *custom-marker.ts* within our park-map-page directory. This new class will extend the base google.mapsMarker to have one additional value, our parkData. We first need to import the Park interface. Then we will export our new class, CustomMapMarker that is extended from google.maps.Marker. Next, we define our parkData variable and assign the type of Park. Within the class' constructor, we will pass in the actual park data. The critical bit of code is the super(). This will tell the class we extended from to also initialize.

```
import {Park} from '../../interfaces/park';

export class CustomMapMarker extends google.maps.Marker{
  parkData:Park
  constructor( theParkData:Park
  ){
    super();
    this.parkData = theParkData;
  }
}
```

Save this file, and return back to *park-map-page.ts*. If you guessed that we need to import this new class, you would be correct.

```
import {CustomMapMarker} from './custom-marker';
```

Now, our parkMarker can use our CustomMapMaker class in place of the google.maps.Marker. So this line of code:

```
let parkMarker:google.maps.Marker = new google.maps.Marker();
```

becomes

```
let parkMarker:google.maps.Marker = new CustomMapMarker(thePark);
```

Note, we are passing the park's data into the instance, thus saving our park data within each marker.

Now, we can assign our event listener for each marker. But how do we reference the actual parkdata stored within each marker, so we can include it as a navParam?

We are going to take a shortcut with this block of code. Since we did not define an interface for our CustomMapMarker, our compiler does not know about our additional property. But, we can use the **any** data type to sidestep this issue. So, if we simply create a local variable, selectedMarker with the type of any and assign the parkMarker to it, we will be able to reference the parkData. Here is the completed fragment:

```
google.maps.event.addListener(parkMarker, 'click', () => {
  let selectedMarker:any = parkMarker;

  this.nav.push(ParkDetailsPage, {
    parkData: selectedMarker.parkData
  });
});
```

The navigation code should look very familiar from the Park List Page. Here is the complete initializeMap function:

```
initializeMap() {
  let minZoomLevel:number = 3;

  this.map = new google.maps.Map(document.getElementById('map_canvas'), {
    zoom: minZoomLevel,
    center: new google.maps.LatLng(39.833, -98.583),
    mapTypeControl: false,
    streetViewControl: false,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  });

  let image:string = 'img/nps_arrowhead.png';

  this.parkData.getParks().then(theResult => {
    this.parks = theResult;

    for (let thePark of this.parks) {
      let parkPos:google.maps.LatLng =↵
          new google.maps.LatLng (thePark.lat, thePark.long);
      let parkMarker:google.maps.Marker = new CustomMapMarker(thePark);
```

```
        parkMarker.setPosition(parkPos);
        parkMarker.setMap( this.map);
        parkMarker.setIcon(image);

        google.maps.event.addListener(parkMarker, 'click', () => {
          let selectedMarker:any = parkMarker;

          this.navCtrl.push(ParkDetailsPage, {
            parkData: selectedMarker.parkData
          });
        });
      }
    })
  }
```

Save the file, and we should now be able to click on a marker and see the Park Details page.

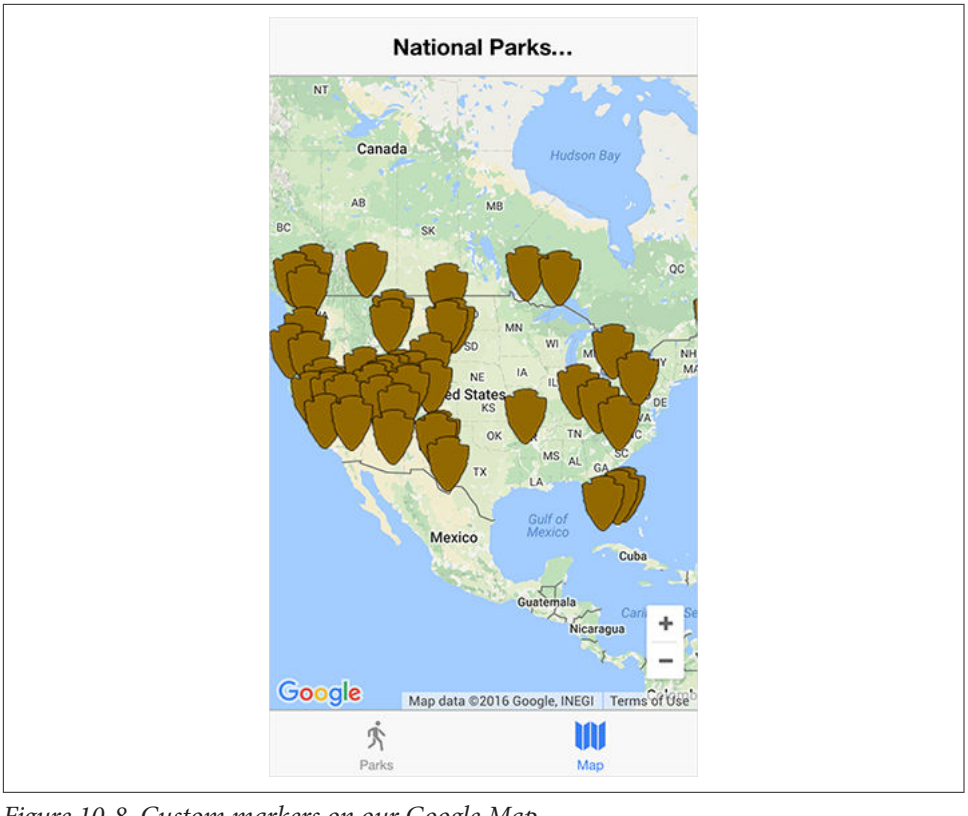


Figure 10-8. Custom markers on our Google Map

Adding Search

Let's extend our application a bit further by adding a search bar for the Park List. Ionic has an **<ion-searchbar>** as part of the component library. The search bar component will let the user type in the name of a National Park and the list parks will automatically update.

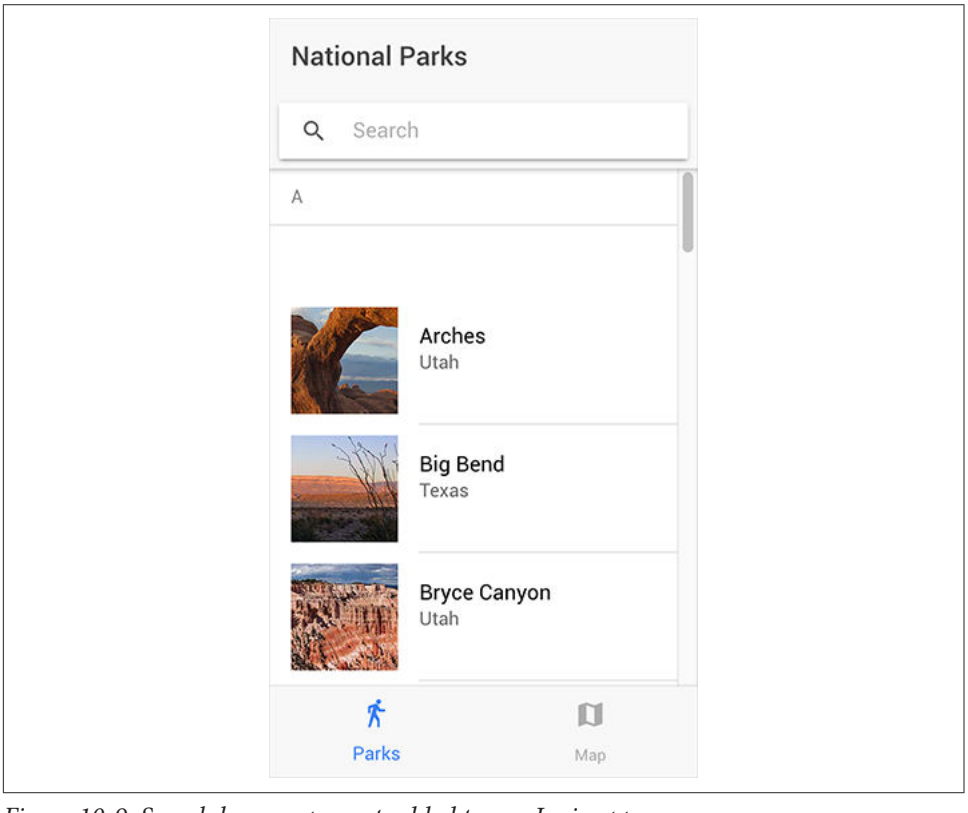


Figure 10-9. Search bar component added to our Ionic app.

Since we want the search bar to always be available, we need to be fixed to the top of the screen. We can use the **<ion-toolbar>** component to handle this. This component just needs to be after the **<ion-navbar>**.

We'll need to define a model to our <ion-searchbar> component and bind it to the query string. Also, we need to add a function to handle user input.

```
<ion-toolbar>
  <ion-searchbar [(ngModel)]="searchQuery" (ionInput)="getParks($event)" ↵
                 (ionClear)="resetList($event)">↵
  </ion-searchbar>
</ion-toolbar>
```

If you are wondering what the [( )] is doing around ngModel, this is a new syntax for Angular's two-way data-binding. The square brackets tell Angular that this is a getter,

while the parentheses tell Angular that this is a setter. Putting the two together, you have two-way data-binding.

Now in the *park-list.ts* file, we need to add our getParks function:

```
getParks(event) {
  // Reset items back to all of the items
  this.parkData.getParks().then(theResult => {
    this.parks = theResult;
  })

  // set queryString to the value of the searchbar
  let queryString = event.target.value;

  if (queryString !== undefined) {
    // if the value is an empty string don't filter the items
    if (queryString.trim() == '') {
      return;
    }

    this.parkData.getFilteredParks(queryString).then(theResult => {
      this.parks = theResult;
    })
  }
}
```

This the first part of the getParks function ensures that we will be using the original list of parks. If you have coded any filtering functions in the past, you are probably aware that you need to make sure that you are working from an unfiltered list.

Next, we get the query string from the search bar, then check that it is not undefined nor empty.

Finally, we will call a new method on the parkData provider (*park-data.ts*) to do the actual filtering based on the search string, and set the results to the parks.

```
getFilteredParks(queryString) {
  return this.load().then(Parks => {
    let theFilteredParks: any = [];

    for (let thePark of Parks) {
      if (thePark.name.toLowerCase().indexOf(queryString.toLowerCase()) > -1) {
        theFilteredParks.push(thePark);
      }
    }

    return theFilteredParks;
  });
}
```

We first make sure we have the master park data again, then we define a new empty array that we will push any matching parks onto it. The code then loops through each

park and compares the park's name against the query string. This code does take an additional step and forcing both the park name and query string to lowercase before testing if it can find a match. If a match is found, it is pushed to theFilteredParks array. Once all the parks have been examined, this array is returned and our displayed list automatically updated.

Our search is still not quite functionally complete. The clear button is not working. Although we bound the ionClear event to a resetList function, we haven't written it yet. The function is actually quite simple, we just need to reset our parks array back to the full list.

```
resetList(event) {
  // Reset items back to all of the items
  this.parkData.getParks().then(theResult => {
    this.parks = theResult;
  })
}
```

Wth that, we should have a fully functioning search bar in our app.

Theming our application

Now that, we have a functioning app it certainly could use a touch of color to brighten it up from the default look. There is nothing special about styling an Ionic-based application. The same techniques used in styling a web app or web page apply here.

Ionic uses Sass or Syntactically Awesome Style Sheets as the CSS pre-processor solution. If you have not used a CSS pre-processor before, one of the main reasons to do so is it gives you the ability to abstract your CSS in reusable parts. For example, you can define a variable that is the app's primary color, and let the preprocessor apply it throughout the CSS. So, if you need to change it for some reason, you change it one place.

Ionic breaks down their Sass files into two file; **src/app/app.scss** for any global app specific styling and **src/theme/variables.scss** for the predefined theme colors.

The first styling change to make is to assign a forest green color to the header. There are several ways accomplish this: we could directly style each specific component, or modify one of the prebuilt themes. For these components, let's choose the latter solution.

In *variables.scss* file, replace the hex color associated with the primary color with #5a712d. Since we did not assign a theme to either the header nor the tabs components, we need to do so. In each of the three pages, update the `<ion-navbar>` to `<ion-navbar color="primary">`.

In tabs.html, replace `<ion-tabs>` with `<ion-tabs color="primary">`. Saving all these files, and run `$ ionic serve`, the headers and tabs should now have a forest green background.

Now, let's turn our attention to styling the various content elements in the app. Let's change the general `<ion-content>` background color to a light tan. In the *app.scss* file, add the following CSS:

```
ion-content {background-color: #f4efdd;}
ion-card-header {background-color: #cfcbbb; font-weight: bold;}
ion-card-content{margin-top: 1em;}
ion-item-divider.item {background-color: #ab903c; color: #fff;
     font-size: 1.8rem; font-weight: bold !important;}
.item {background-color: #f4efdd;}
.toolbar-background {background-color: #706d61;}
.searchbar-input-container {background-color: #fff;}
```

As you can see this CSS is a mix of styling the Ionic components directly, such as the ion-content, or ion-card-header, but also setting specific CSS classes. By setting this in the *app.scss*, these will be applied throughout the app. If you needed to set the style of a specific page or component, then you would do that within the .scss file for that item. Let's do that now.

The park's name on the park detail's page is a bit too far down from the header photo. Open the park-details.scss file and change it to

```
page-park-details {
  h1{margin-top: 0;}
}
```

Now the <h1> tag only on the park details page will have it's top margin set to zero, leaving any other <h1> tags we might use styled as they normally would be.

The Ionic framework actually exposes quite a few variables in their SCSS that you can override. You can find a full list at http://ionicframework.com/docs/v2/theming/overriding-ionic-variables/.

If you are ever having trouble applying a style, remember this is all just CSS, and the web inspector in your browser can assist in finding either the target you need to address or uncover a cascading issue.

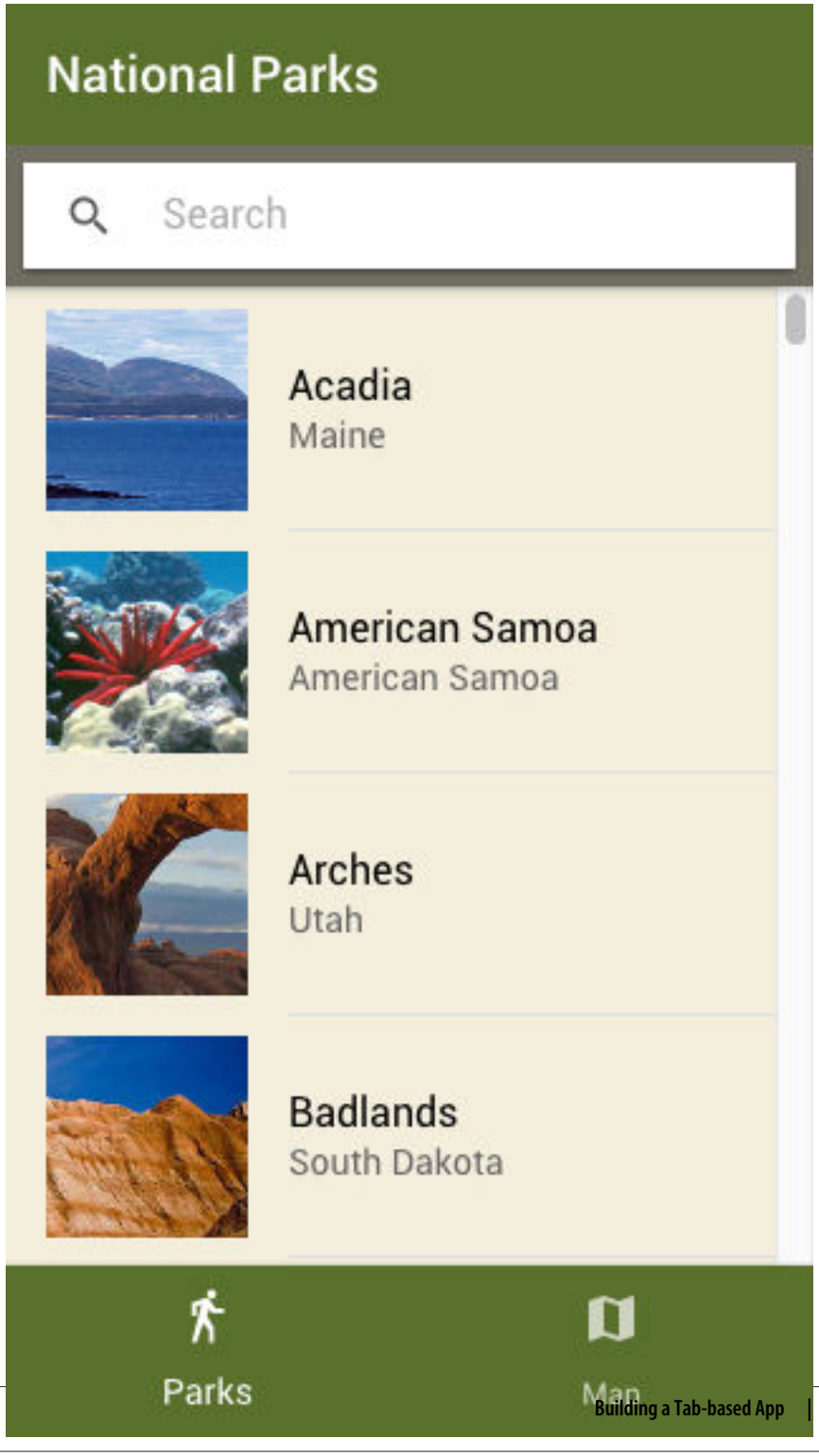


Figure 10-10. Our styled National Parks app.

Virtual Scrolling

One of greatest impacts on the performance of a hybrid application is the number of DOM elements on a given page. Usually, this manifests itself as an issue when scrolling lists of content. This is typically how most users can spot a hybrid application by the poor scrolling performance. Knowing this, the team at Ionic focused their efforts to create as native as possible scrolling experience.

Our app only has 59 items, and each row is rather straight-forward in the elements it contains. If we expanded our app to include all the National Monuments, our list would exceed 400. At this value, we probably would start to see some stutter and jerkiness in the scrolling.

To address this issue, the Ionic framework introduced a special set of directives, known collectively as Virtual Scrolling.

What about Collection Repeat?

If you used Ionic 1.x, you might be familiar with Collection Repeat. This was the system that was introduced to solve the scrolling large dataset problem. It has been replaced with the Virtual Scrolling solution.

Instead of creating DOM elements for every item in the list, only a small subset of records (enough to fill the viewport) are rendered and reused as the user scrolls. The Virtual Scroller manages this completely behind the scenes for us.

There are a few differences between a standard list and one that uses Virtual Scrolling. First, the <ion-list> now has [virtualScroll] binding to our data. The data given to the virtualScroll property must be an array. Second, the <ion-item> now has a *virtualItem property, which references the individual item that will be passed into the template.

It is probably easier if you see the revised code for a park's list:

```
<ion-list [virtualScroll]="parks">
  <ion-item *virtualItem="let park" (click)="goParkDetails(park)" detail-push>
      <ion-thumbnail item-left>
          <img src="assets/img/thumbs/{{park.image}}">
      </ion-thumbnail>
      <h2>{{park.name}}</h2>
      <p>{{park.state}}</p>
  </ion-item>
</ion-list>
```

Other, then replacing the *ngFor="let park of parks", with the two virtualScroll properties, the code remains the same. However, there is one more change we should

make to improve our list's scrolling performance, and that is to replace the <img> tag with <ion-img>.

```
<ion-img src="assets/img/thumbs/{{park.image}}"></ion-img>
```

This tag designed for use specifically with the virtual scrolling system. The <ion-img> tag manages HTTP requests and image rendering. Additionally, it includes a customizable placeholder element which shows before the image has finished loading. While scrolling through items quickly, <ion-img> knows not to make any image requests, and only loads the images that are viewable after scrolling.

Some additional performance tweaks that you might consider making to improve performance are

- Image sizes should be locked in, meaning the size of any element should not change after the image has loaded.
- Provide an approximate width and height so the virtual scroll can best calculate the cell height.
- Changing the dataset requires the entire virtual scroll to be reset, which is an expensive operation and should be avoided if possible.

Custom List Headers

The virtual scrolling system also supports dynamic headers and footers. In this sample, our list will have a header inserted after every 20th record.

```
<ion-list [virtualScroll]="items" [headerFn]="customHeaderFn">

  <ion-item-divider *virtualHeader="let header">
    {{ header }}
  </ion-item-divider>

  <ion-item *virtualItem="let item">
    Item: {{ item }}
  </ion-item>

</ion-list>
```

and the supporting function would be:

```
customHeaderFn(record, recordIndex, records) {
  if (recordIndex % 20 === 0) {
    return 'Header ' + recordIndex;
  }
  return null;
}
```

When applied this to our list of National Parks, our <ion-list> becomes

```
<ion-list [virtualScroll]="parks" [headerFn]="customHeaderFn">
  <ion-item-divider *virtualHeader="let header">
    {{ header }}
```

```
    </ion-item-divider>

    <ion-item *virtualItem="let park" (click)="goParkDetails(park)" detail-push>
      <ion-thumbnail item-left>
        <!--<img src="assets/img/thumbs/{{park.image}}">-->
        <ion-img src="assets/img/thumbs/{{park.image}}"></ion-img>
      </ion-thumbnail>
      <h2>{{park.name}}</h2>
      <p>{{park.state}}</p>
       </ion-item>
  </ion-list>
```

and in our *park-list.ts* file, we can add the following function to insert the first letter of the park's name into our custom header:

```
customHeaderFn(record, recordIndex, records) {
  if ( recordIndex > 0) {
    if ( record.name.charAt(0) !== records[recordIndex-1].name.charAt(0)) {
      return record.name.charAt(0);
    } else {
      return null;
    }
  } else {
    return record.name.charAt(0);
  }
}
```

One last piece, will be to provide a little bit of styling on ion-item-divider. Make the change in the *park-list.scss* to:

```
.parklistPage {
    ion-item-divider {
        background-color: #ad8e40;
        font-weight: bold;
    }
}
```

So, we now have an app that looks like this:

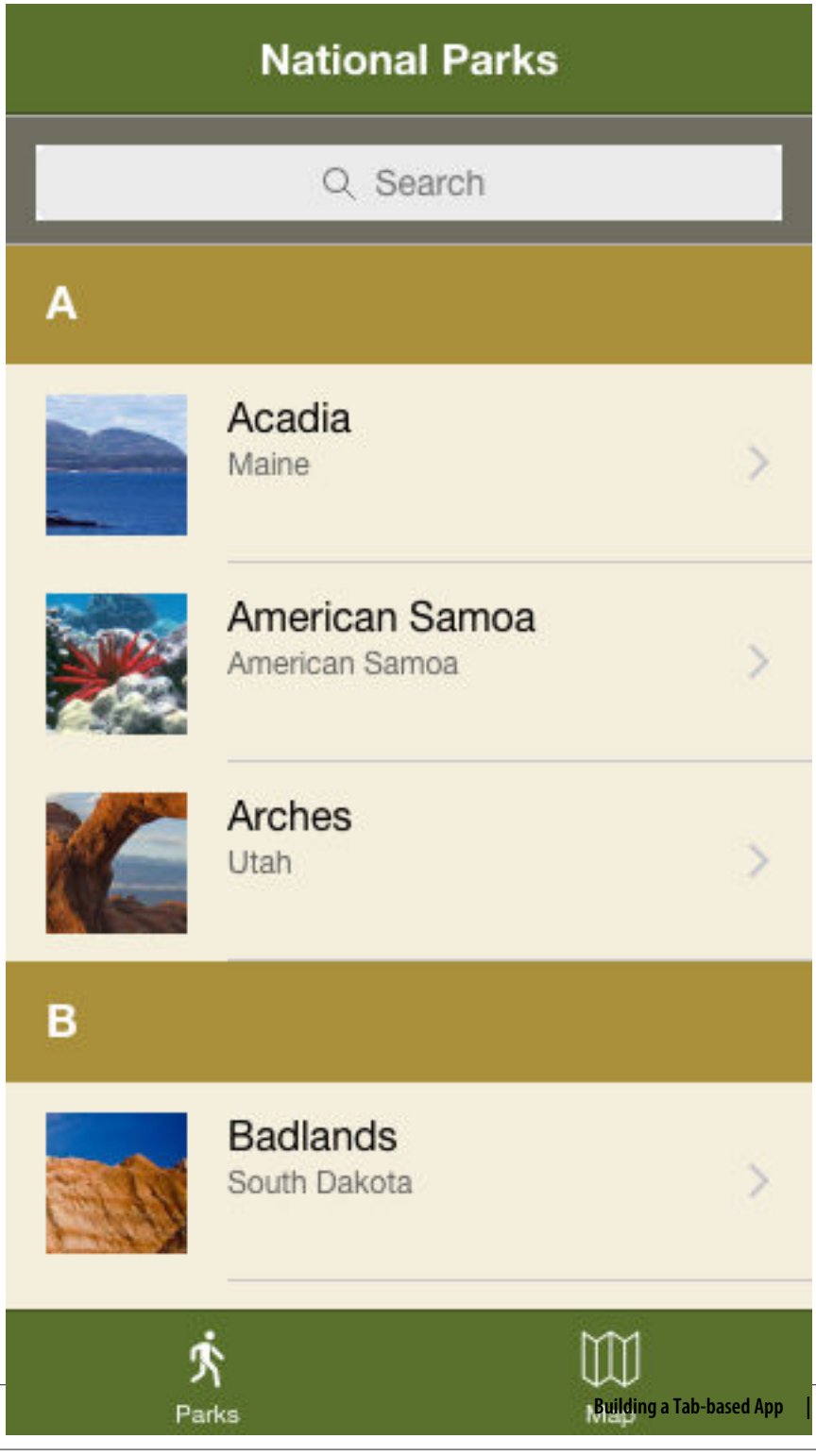


Figure 10-11. Virtual Scrolling with dynamic headers applied.

Summary

With this app we have explored how to work with a tab-based design, used a data provider, integrated a Google Map and applied some basic theming. If you want to extend this app some, here are a couple of ideas that you can try. First, add a park specific map to the page details screen. Second, look at adding a photo slide show to each park. Ionic has a **<ion-slides>** component that can do the trick. Finally, if you want a real challenge, look at calculating the distance to each park. There is a distance property already in the data.json file. You can leverage the Geolocation plugin to find your current latitude and longitude, then use the haversine formula to calculate the distance.

CHAPTER 11
Building a Weather Application

One of other prebuilt templates that the Ionic framework provides is the sidemenu template. This design pattern has become increasingly prevalent over the past few years. Rather than having a fixed number of items in a tab bar (which typically remain on-screen using precious screen real estate), this user interface moves many of the navigation options onto a panel that is kept off-screen until the user taps on a menu button of some kind. Often this button is either three horizontally stacked lines (aka the hamburger menu) or in some cases three vertical dots. Now, I am not going to get into the pros and cons of this user interface element. I would encourage you to spend a little time researching it on your own to see if it is right for your project. With that said, this is the design pattern we will use to build our IonicWeather app.

Getting Started

Like the other two projects before, we need to generate our initial app. We use another starter template from a GitHub repo. This base template is just the sidemenu template, with some additional elements in the assets folder.

```
$ ionic start Ionic2Weather https://github.com/chrisgriffith/Ionic2Weather --v2
```

Once this process is complete, again remember to change your working directory:

```
$ cd Ionic2Weather
```

and if you targeting Android, don't forget to add that platform:

```
$ ionic platform add android
```

Let's take a look at the template in our browser with

```
$ ionic serve
```

Here is what you should see in your browser:

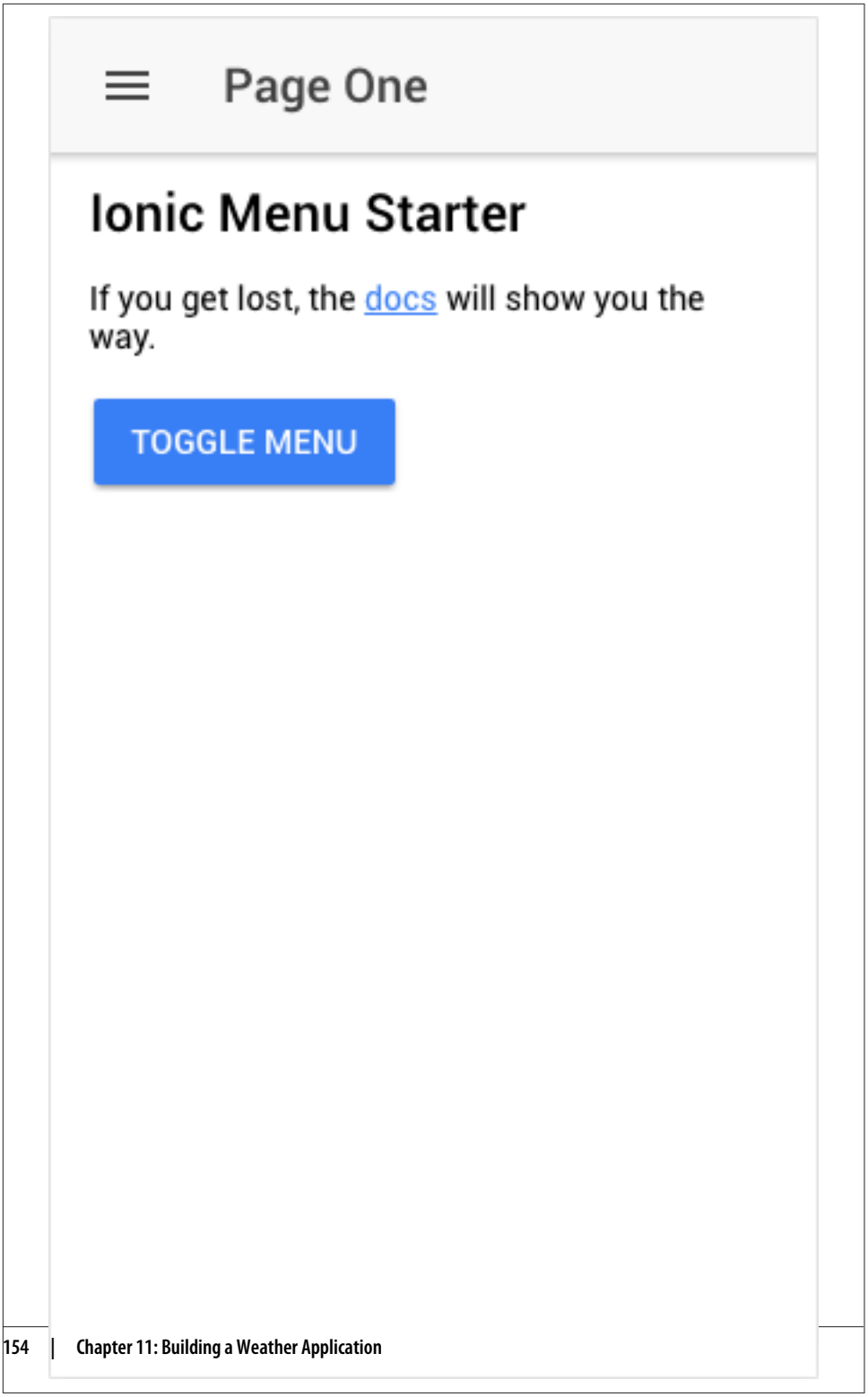


Figure 11-1. The starter sidemenu template

The template is fairly straight forward. The main screen demonstrates two methods to reveal the sidemenu; either by the menu icon in the navbar or by the button in the content section. Once you tap on either item, the menu animates over the screen, and the existing content is masked by a semi-transparent overlay.

The sidemenu itself contains two sections: its own navbar, here labeled Menu and its own content, here showing a list of two items (Page One and Page Two). If you tap outside the sidemenu, Ionic will automatically dismiss the sidemenu for you. If you tap on either the Page One or Page Two, the main content of our app will update, and the sidemenu dismissed.

Exploring the Sidemenu template

The sidemenu template is a bit more complex that the other two templates, so let's take a long look at the base template before we build our IonicWeather app. There are no changes to the default index.html file, so there is no need to explore it. Instead, we will start with the *app.html* file. This file can be found in the src/app directory. This is our initial HTML that is rendered by our app. Unlike the other two templates, this template's app component references an external HTML template (app.html) instead of having the template written inline. As we'll see, since this template is a bit more complex than the blank or tab templates so it makes sense to have it as an external reference. Here is what the template looks like:

```
<ion-menu [content]="content">
  <ion-header>
    <ion-toolbar>
      <ion-title>Menu</ion-title>
    </ion-toolbar>
  </ion-header>

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

</ion-menu>

<!-- Disable swipe-to-go-back because it's poor UX to combine STGB with side menus -->
<ion-nav [root]="rootPage" #content swipeBackEnabled="false"></ion-nav>
```

There are quite a few changes from Ionic 1's sidemenu structure. Most notably `<ion-side-menus>` and `<ion-side-menu-content>` tags have been removed and replaced with `<ion-menu>` and the new page reference system.

Let's look at the `<ion-menu>` tag in detail. This tag sets up the content that will be displayed within our side menu. But there is a critical attribute that is also set within this

tag, the [content]="content". Here we are setting the **content** property of the ion-menu to the variable **content** and not a string. If you look at the last tag in the template you will see a reference to #content, which defines it as a local variable. Now our <ion-menu> can reference it properly and use it for its main content. We will come back to the rest of the attributes <ion-nav> in a bit.

Next, our template defines the header of the sidemenu by using the <ion-toolbar> tag. This component is a generic bar that can be placed above or below the content, much like the <ion-navbar>, but without the navigation controls.

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

Then, we define the sidemenu content in the <ion-content> tag.

```
<ion-content>
  <ion-list>
    <button menuClose ion-item *ngFor="let p of pages" (click)="openPage(p)">
      {{p.title}}
    </button>
  </ion-list>
</ion-content>
```

The template uses a <ion-list> to loop through the pages array and create a list of buttons that are labeled with the object's title property. It also sets up a click handler, openPage to respond to the user tapping on the button.

Now, let's return to the <ion-nav> tag and explore it in more detail

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

The root property is set to rootPage. This is defined in the *app.component.ts* file, and will be the property we will update when we need to change our main content. Now, let's turn our attention to the *app.component.ts* file and how these elements are linked together in code.

Exploring the app.component.ts file

Just like the sidemenu's HTML template was a bit more complex, so is its *app.ts* file. First, we have several more import statements than before.

```
import { Component, ViewChild } from '@angular/core';
import { Nav, Platform } from 'ionic-angular';
import { StatusBar } from 'ionic-native';
import { Page1 } from '../pages/page1/page1';
import { Page2 } from '../pages/page2/page2';
```

In addition to the Component module being imports, we are also importing the ViewChild component from @angular/core. Including this import enables our code to use the property decorator **@ViewChild** to define a reference to a child element.

This is the underlying mechanism that allows us to properly reference our nav element.

The next two import statements should look fairly familiar to you. We need the Nav and Platform directives imported from Ionic and the reference to the StatusBar from Ionic Native.

The final two imports are the two sample content pages in the template. Since our *app.component.ts* handled the sidemenu navigation, it needs to have a reference to any screen it may navigate to.

Our @Component decorator should also look familiar. As pointed out earlier, it references an external template.

Next, we define our rootPage variable to be of data type **any**, and set it initially to the Page1 component.

```
rootPage: any = Page1;
```

For the sidemenu's list of pages, that array is defined,as is the structure of its contents.

```
pages: Array<{title: string, component: any}>
```

Here is a great example of how TypeScript can provide some code safety by defining the element types.

The constructor then initializes the app (which for the moment handles a status bar call on iOS, once the platform is ready), and sets the pages array to our two sample pages.

Finally, we define the openPages function that is called from the sidemenu buttons.

```
openPage(page) {
  // Reset the content nav to have just this page
  // we wouldn't want the back button to show in this scenario
  this.nav.setRoot(page.component);
}
```

This function takes the page object that is passed into the function and then tells the nav controller to set its root property to the page.component. By setting the navigation root, we reset the navigation stack history. In doing so, we also prevent Ionic from automatically adding a back button.

Side Menu Options

You might have noticed we actually have not set any options for our sidemenu.

The menu can be placed either on the left (default) or the right. To change the placement of the sidemenu, set the side property to the position you want.

```
<ion-menu side="right" [content]="content">...</ion-menu>
```

The sidemenu supports three display types: **overlay**, **reveal**, and **push**. The Ionic framework will use the correct style based on the currently running platform. The default type for both Material Design and Windows mode is overlay, and reveal is the default type for iOS mode.

If you want to change it directly on the `<ion-menu>` tag, use this

```
<ion-menu type="overlay" [content]="content">...</ion-menu>
```

However, you can set the sidemenu type globally via the app config object. This is set in the *app.module.ts* file. The sample below sets the base style to push, and then sets the menuType to reveal for md mode.

```
imports: [
  IonicModule.forRoot(MyApp, { menuType: 'push',
  platforms: {
   md: {
     menuType: 'reveal',
   }
  }})
]
```

Displaying the Menu

To actually display our side menu, we can use the menuToggle directive anywhere in our template. Let's look at the *page1.html* file (src/pages/page1), and you will find the HTML block that defines the `<ion-navbar>`

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Page One</ion-title>
  </ion-navbar>
</ion-header>
```

On the `<button>` element, you will see the **menuToggle** directive has been applied. This is all we need to do in order for Ionic to display our side menu. This does not define the visible element for our sidemenu button. That is done in this example by the `<ion-icon>` and setting the name to **menu**.

In fact, if you look further in the page1.html code, you will see this

```
<button ion-button secondary menuToggle>Toggle Menu</button>
```

Here, the button element has the **menuToggle** directive applied to it, thus allowing it to open the sidemenu as well. Although the Ionic framework handles a modest amount of work for us in the opening and closing the sidemenu, but there may be times when we need to directly close the sidemenu. We add this functionality by using the menuClose directive on our element.

```
<button ion-button menuClose="left">Close Side Menu</button>
```

That covers most of the basics of the sidemenu template. Let's get started building our actual application.

Converting the template

Before we being converting the template to our needs, let's generate our two pages that our app will use, as well as the data providers.

```
$ ionic g page weather

$ ionic g page locations
```

Next, let's add in our two providers. The first provider will eventually be the provider that will pull live weather data from darksky.net (formerly Forecast.io) .

```
$ ionic g provider WeatherService
```

Once the command has finished generating the template provider, we need to create our second provider. This provider will be used to take a human-friendly location, like San Diego, and turn it into a corresponding latitude and longitude required by darksky.net. We will touch on both of these providers later in this chapter.

```
$ ionic g provider GeocodeService
```

Now, with our new base pages and providers in place, let's convert this template to use them. Open *app.module.ts* in your editor.

In the import section, we can remove the reference to Page1 and Page 2 and add in the imports to our two newly created pages.

```
import { Weather } from '../pages/weather/weather';
import { Locations } from '../pages/locations/locations';
```

Also, update the declarations and entryComponents arrays to reflect these new pages.

After those imports, we can import our two providers as well:

```
import { WeatherService } from '../providers/weather-service';
import { GeocodeService } from '../providers/geocode-service';
```

In the providers array, we will need to add our two providers as well.

```
providers: [WeatherService, GeocodeService]
```

With those changes made, we can turn to *app.component.ts* and begin to update it. First, replace the two Page imports with our new pages.

```
import { Weather } from '../pages/weather/weather';
import { Locations } from '../pages/locations/locations';
```

Continuing further down in the file, we will see the line of code that defines the root-page variable for our application. We need to change this to reference our Weather instead of Page1.

```
rootPage: any = Weather;
```

Our next bit of code to modify is the pages array. We are going to be extending this array quite a bit over the course of building our app, but for now, we will do just the basics.

Since this app is written in TypeScript, we need to be careful when modifying templates that we properly update any variable definitions. For example, the pages variable is defined to be an array that contains objects with a title property that is of type string, and component property that is of type any. Since we would like to display an icon next to each item in my sidemenu list, we will need to update the definition to:

```
pages: Array<{title: string, component: any, icon: string}>
```

Let's update the pages array to reference the proper component, as well as adding an icon value and changing the title. Here is the new array:

```
this.pages = [
 { title: 'Edit Locations', component: Locations, icon: 'create' },
 { title: 'Current Location', component: Weather, icon: 'pin' }
];
```

Later, we will extend this array to hold our saved locations and their latitudes and longitudes.

If you ran the app in its current form, you would only see the weather page and no ability to access the sidemenu. Let's address that issue now, as well as make a few other changes to the navbar.

Open *weather.html* in your editor.

First, We need to add in the sidemenu button within the <ion-navbar>

```
<button ion-button menuToggle>
  <ion-icon name="menu"></ion-icon>
</button>
```

Next, change the `<ion-title>` to display 'Current Location'. Once we have geolocation enabled, we will return to this tag later and have it display the actual location name.

```
<ion-title>Current Location</ion-title>
```

If you save the file now, you should see the menu icon on the left, and the ability to display our sidemenu. Let's update the header in the locations.html as well:

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Edit Locations</ion-title>
```

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

Now, with that code added, this page will be able to display the sidemenu as well. Switch back to the *app.html* file and we update the code to show our icons next to each list item. We just need to add a `<ion-icon>` tag and set its name to the icon property of the list element. We will also need to add some padding to the right of the icon so that the text is not directly next to the icon itself. The quickest way is to just add a space between the end tag </ion-icon> and the opening mustache tags of {{p.title}} like you see below:

```
<ion-list>
  <button menuClose ion-item *ngFor="let p of pages" (click)="openPage(p)">
    <ion-icon name="{{p.icon}}"></ion-icon> {{p.title}}
  </button>
</ion-list>
```

Another minor tweak is to set the `<ion-title>` to

`<ion-title>Ionic Weather</ion-title>` instead of the default 'Menu'.

Our final tweak is to add a reference to the where we are getting the weather data from. After the <ion-list>, add this bit of markup:

```
<p><a href="ttps://darksky.net/poweredby/">Powered by Dark Sky</a></p>
```

Save this file, and our side menu should now show our icons and have the ability navigate to our two pages.

Mocking up our Weather Provider

Our next step is to get some weather data that we can use in our application. We are going to approach this is two phases. First, we are going to load in static weather data. This will allow us to do some initial screen layout first before we hook into a live data source.

There is a static data file that you can use. You can download this sample JSON file from **xyz.com**. This file is a snapshot of the darksky.net weather data. Now, create a directory named 'data' within the www folder and copy this data file there.

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

@Injectable()
export class WeatherService {
  data: any = null;

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

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

    return new Promise(resolve => {
      this.http.get('assets/data/data.json')
        .map(res => res.json())
        .subscribe(data => {
          this.data = data;
          resolve(this.data);
        });
    });
  }
}
```

One other change to the provider is to add in the method to get the data with. After the load method, add in this function:

```
getWeather() {
  return this.load().then(data => {
    return data;
  });
}
```

Although the method does call the load function, I like having it as a separate method. This gives us some flexibility later.

Now, we just need to have the app tell this provider to load the data. Back in the *app.component.ts* file, we will need to make a few changes to call our provider.

First, we need to import this provider into the component.

```
import { WeatherService } from '../providers/weather-service';
```

In the constructor we need to pass in reference our WeatherData provider:

```
constructor(publice platform: Platform,
public weatherData: WeatherService) {...}
```

Now within the initializeApp function we can call this service and load our weather data:

```
initializeApp() {
  this.platform.ready().then(() => {
    // Okay, so the platform is ready and our plugins are available.
    // Here you can do any higher level native things you might need.
    StatusBar.styleDefault();
  });
 this.weatherData.load();
}
```

With our data now available, let's work on actually displaying it.

Laying Out the Weather Data

Before we can display our mock weather data, we need to have access to it within the context of that page. Open the *weather.ts* file. The first thing we will need to do is to import the WeatherData provider.

```
import { WeatherService } from '../../providers/weather-service';
```

Next, we need to add three variables to hold our weather data with the class definition. We will type these as any, so it will not generate any TypeScript errors.

```
theWeather: any = {};
currentData: any = {};
daily: any = {};
```

The reason there are currentData and daily variables is the actual JSON data structure from darksky.net is a bit nested. So rather than having to traverse this in our template, we can define the currentData and daily variables as pointers.

Next, pass in a reference to our WeatherService into our constructor:

```
constructor(public navCtrl: NavController, public weatherData: WeatherService) { }
```

Now, within our constructor we can call the getWeather method on the weatherData provider:

```
this.weatherData.getWeather().then(theResult => {
  this.theWeather = theResult;
  this.currentData = this.theWeather.currently;
  this.daily = this.theWeather.daily;
});
```

With the data now loaded, let's turn our attention to the HTML template. Open weather.html again so we can begin updating the template to show our weather data.

For the layout of our data, we are going to use the Ionic Grid component. From the Ionic documentation:

> Ionic's grid system is based on flexbox, a CSS feature supported by all devices that Ionic supports. The grid is composed of three units — grid, rows and columns. Columns will expand to fill their row, and will resize to fit additional columns.

To begin, we will add a `<ion-grid>` tag. Next, we will define our first row that will contain the current temperature and conditions.

```
<ion-row>
  <ion-col width-100>
    <h1>{{currentData.temperature | number:'.0-0'}}&deg;</h1>
    <p>{{currentData.summary}}</p>
  </ion-col>
</ion-row>
```

If you take a look at the actual *data.json* file, you will see the temperature values are actual saved out to two decimal places. I doubt we need this level of accuracy for our

app. To solve this, we can use on the of the built in Pipe functions in Angular to round our values for us.

What is a Pipe?

A pipe takes in data as input and transforms it into the desired output. Angular includes several stock pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe.

By adding **| number:'.0-0'** after the data binding for currentData.temperature, our data will go from 58.59 to 59.

Next, we will add another row that will display the high and low temperatures for the next three days:

```
<ion-row>
  <ion-col width-33>
    {{daily.data[0].temperatureMax | number:'.0-0'}}&deg;<br>
    {{daily.data[0].temperatureMin | number:'.0-0'}}&deg;
  </ion-col>
  <ion-col width-33>
    {{daily.data[1].temperatureMax | number:'.0-0'}}&deg;<br>
    {{daily.data[1].temperatureMin | number:'.0-0'}}&deg;
  </ion-col>
  <ion-col width-33>
    {{daily.data[2].temperatureMax | number:'.0-0'}}&deg;<br>
    {{daily.data[2].temperatureMin | number:'.0-0'}}&deg;
  </ion-col>
</ion-row>
```

Now, is you save the file and try to run the app, you will encounter an error. This error occurs because our data is being loaded via a Promise and is not initially available to the template. To fix this issue, we can add a ngIf directive to tell the template not to display our grid until we have our data. Although we could have set our variables to some initial values, and thus allowed the template to render. Any of our non-data bound elements would be shown, like the ° symbol. By wrapping the entire grid in a ngIf, we can control the display of everything. So our `<ion-grid>` tag becomes:

```
<ion-grid *ngIf="daily.data != undefined">
```

Saving the file again, our template will correctly render once the data has been properly loaded.

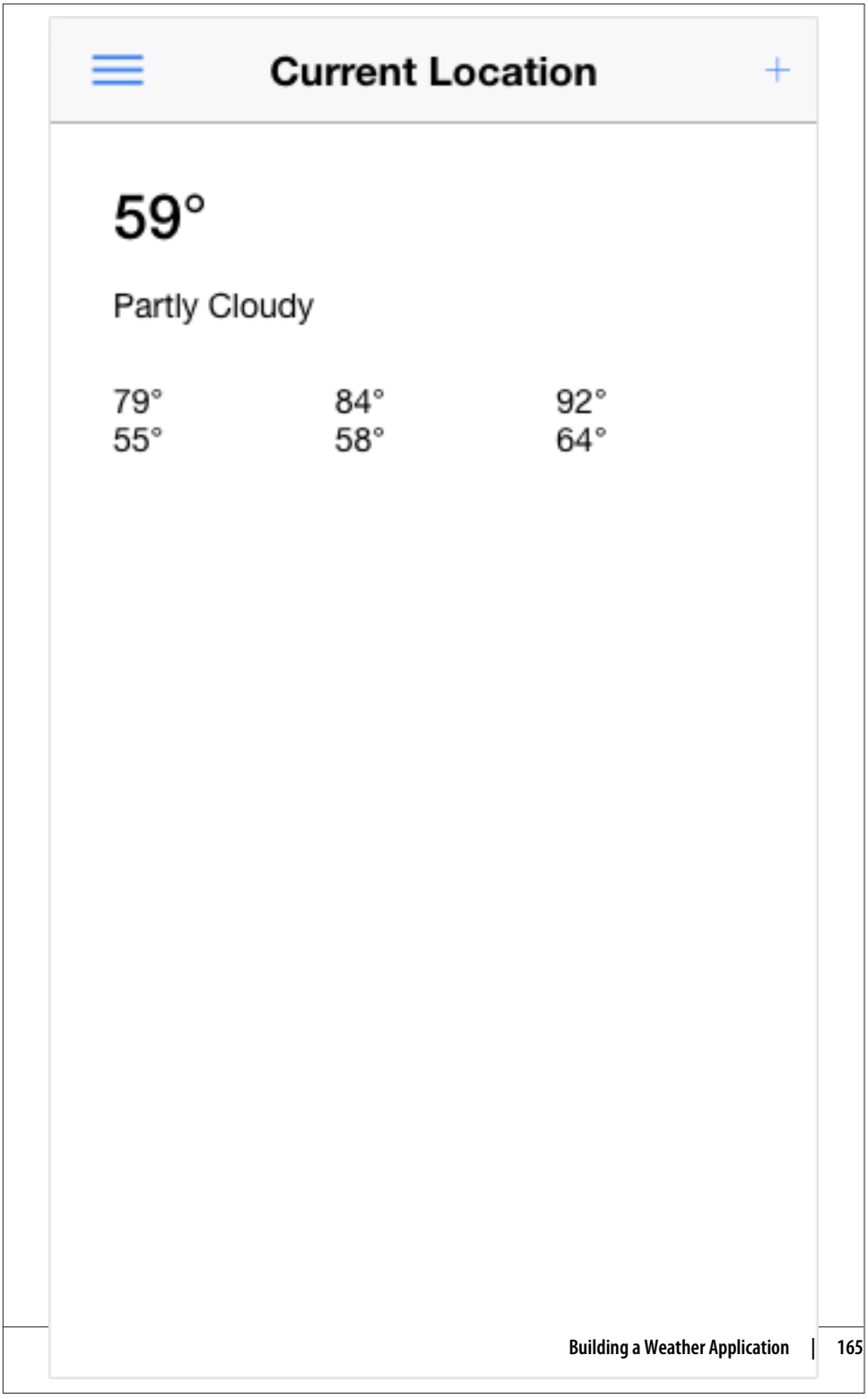


Figure 11-2. Our mock weather data being displayed

Loading feedback: Loading Dialogs and Pull to Refresh

It would be nice if there was some feedback that the app was loading the weather data. Although the load time might be brief when using our mock data, once we hook it up to a live data source, it might take a moment or two.

Ionic has a LoadingController component that we can easily add into our app. Update the import statement for ionic-angular to include Loading.

```
import { NavController, LoadingController } from 'ionic-angular';
```

Next, in the class definition add in a loading variable of typing Loading:

```
export class WeatherPage {
  theWeather: any = {};
  currentData: any = {};
  daily: any = {};
  loading: LoadingController;
```

We will also need to pass in this module into our constructor:

```
constructor(public navCtrl: NavController,
            public weatherData: WeatherService,
            public loadingCtrl: LoadingController) {...
```

Now, we can create our loading dialog component instance. Add this code before making the **weatherData.getWeather()** call.

```
let loader = this.loadingCtrl.create({
  content: "Loading weather data...",
  duration: 3000
});
```

We are setting a duration for the dialog for the moment for testing purposes since our data is still being load locally. To display the loading dialog, we need to simply tell the loader instance to show the dialog via the present method.

```
loader.present();
```

For more information about the Loading component, see http://ionicframework.com/docs/v2/components/#loading

Saving this file, and running the app, the loading dialog will appear for 3 seconds and then dismiss itself.

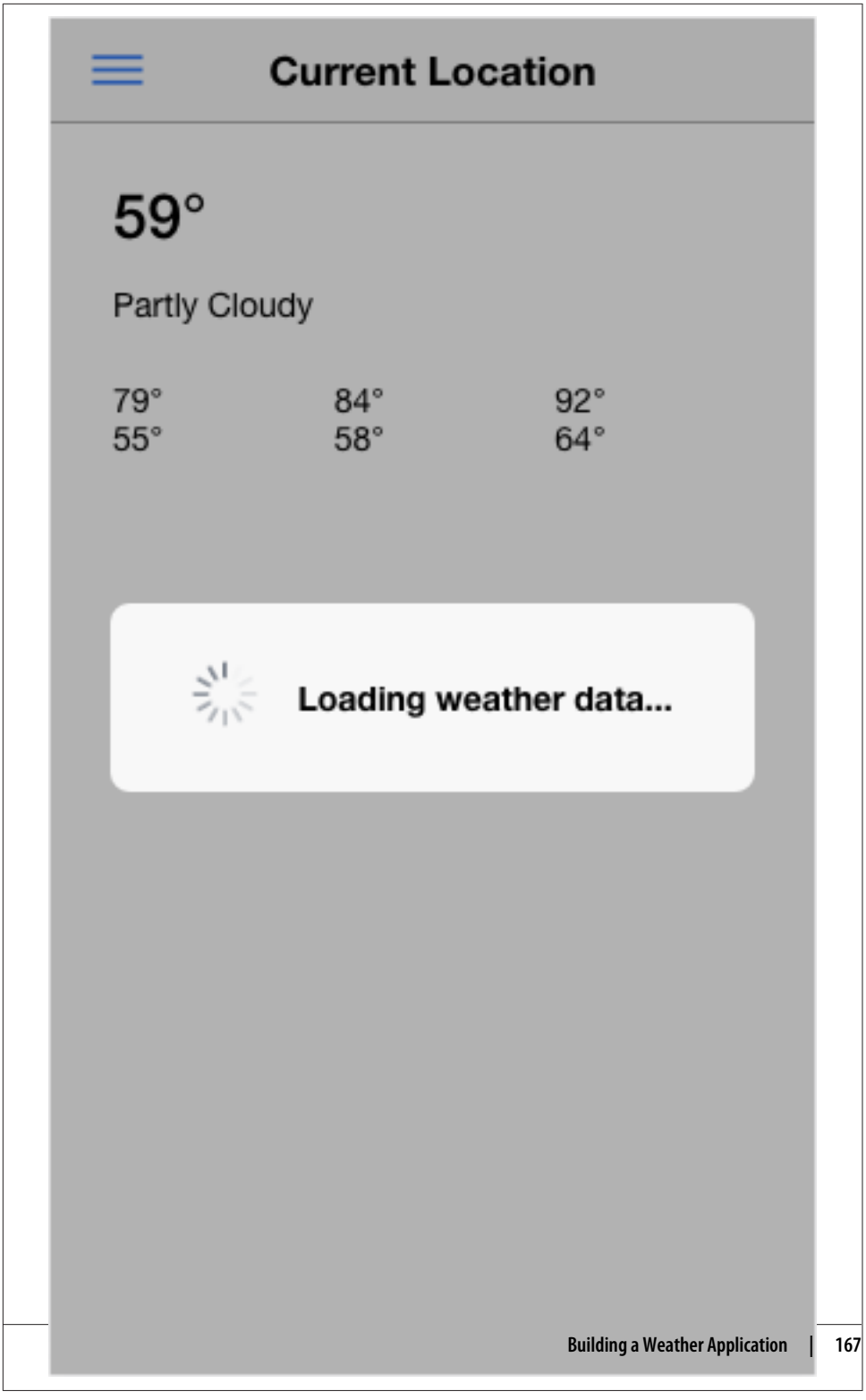


Figure 11-3. The loading dialog

Since we are working with dialogs and data updates, let's add in the code to allow us to use the Refresher component (aka Pull To Refresh). This is a popular UX method to allow users to force a data refresh without having to have an on-screen control to enable it.

Ionic has a prebuilt component, Refresher, for this that we can easily add into our application.

In the *weather.html* file, after the `<ion-content>`, we can add this following:

```
<ion-refresher (ionRefresh)="doRefresh($event)">
  <ion-refresher-content
    pullingIcon="arrow-dropdown"
    pullingText="Pull to refresh"
    refreshingSpinner="circles"
    refreshingText="Refreshing...">
  </ion-refresher-content>
</ion-refresher>
```

This will define the `<ion-refresher>` and `<ion-refresher-content>`. Beyond setting the text that is displayed, you can also define which spinner animation you want. The built-in options are; ios, ios-small, bubbles, circles, crescent, and dots. When the pull to refresh component is triggered it will call the doRefresh method. Let's add in the doRefresh method to the *weather.ts* file.

First, we need to include this component as part of the imports from Ionic:

```
import { NavController, LoadingController, Refresher } from 'ionic-angular';
```

Then include a variable to reference the Refresher in the class definition:

```
export class WeatherPage {
  theWeather: any = {};
  currentData: any = {};
  daily: any = {};
  loading: LoadingController;
  refresher: Refresher;
```

Now, we will add in the doRefresh method to our class:

```
doRefresh(refresher) {
  setTimeout(() => {
    refresher.complete();
  }, 2000);
}
```

For the moment, we are just triggering a simple timeout call for 2 seconds. We will return to this method once we have live data enabled, but this will allow us to see the refresher's UI in action.

Adding GeoLocation

The Dark Sky API requires us to pass in a latitude and longitude value in order to get the weather forecast. We can obtain this set of data through the use of Cordova's Geolocation plugin. To add this plugin to our app, we need to use this command in the terminal.

```
$ ionic plugin add cordova-plugin-geolocation
```

The CLI will add the plugin for all the install platforms. Note, this command only installs the plugin itself, it does not write any of the actual code to use this plugin.

Open *weather.ts*, and first, let's import the Geolocation module from Ionic Native

```
import { Geolocation } from 'ionic-native';
```

If you recall, Ionic Native acts as an Angular wrapper for your Cordova plugins.

Next, in the constructor, we will call the getCurrentPosition function on the Geolocation module. For the moment, it will just write out your current latitude and longitude to the console.

```
Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
});
```

Since making GPS requests are battery consuming, we should save this result and use it instead. Instead of saving this to a generic object, let's define a properly defined variable that implements the CurrentLoc interface.

To do this, we need to define that interface. Create a new directory named interfaces within the app directory. Within that new directory, create a new file named *current-loc.ts*. This file will hold our simple definition for our CurrentLoc interface. The code for this is:

```
export interface CurrentLoc {
  lat: number;
  lon: number;
  timestamp?: number;
}
```

This interface will tell the compiler that it needs both a lat and long property with their allowed values as numbers. It also tells the compiler that an optional property, timestamp can be passed. This is done by adding in the **?** after the property name

Returning back to *weather.ts*, we need to import this interface for use in our class:

```
import { CurrentLoc } from '../../interfaces/current-loc';
```

Now, we can create a currentLoc variable that implements the CurrentLoc interface.

```
currentLoc: CurrentLoc = {lat:0 , lon: 0};
```

Our getCurrentPosition call can now save our result into our currentLoc variable:

```
Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
  this.currentLoc.lat = pos.coords.latitude;
  this.currentLoc.lon = pos.coords.longitude;
  this.currentLoc.timestamp = pos.timestamp;
});
```

We will save the timestamp, as well, in case we want to check the age of the data and trigger a refresh. I will leave that to you as a programming challenge once the app is complete. Now that we know where on this planet you are, let's find out your weather.

Accessing Live Weather Data

There are a variety of weather services you can use, but for this app, we will use the Dark Sky service. They have a developer account that you can sign up for at https://darksky.net/dev/register.

Once you have signed up, you will be issued an API key. We will need this in order to use their API. Currently, their API allows up 1,000 calls per day before you have to pay for usage.

The call to the API is actual quite simple, it is just:

```
https://api.darksky.net/forecast/APIKEY/LATITUDE,LONGITUDE
```

If we just replaced our local data call in the *weather-data.ts* file with this URL (replacing the APIKEY with your API key, and also hard coding a location), you will find that it will not work. Opening the JavaScript console, you will see an error like this:

XMLHttpRequest cannot load https://api.darksky.net/forecast/APIKEY/LATITUDE,LONGITUDE. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:8100' is therefore not allowed access.

This error happens due to the browser's security policies, which essentially a way to block access to data from other domains. It also referred to as CORS (Cross Origin Resource Sharing). Typically solving this issue during development requires, setting up a proxy server, or some other workaround. Thankfully, the Ionic CLI has a built-in workaround.

In the *ionic.config.json* (which can be found at the root level of our app), we can define a set of proxies for ionic serve to use. Just add in a proxies property and define the path and the proxyUrl. In our case it, will look like this:

```
{
  "name": "IonicWeather",
  "app_id": "",
  "v2": true,
  "typescript": true,
  "proxies": [
    {
      "path": "/api/forecast",
```

```
      "proxyUrl": "https://api.darksky.net/forecast/APIKEY"
    }
  ]
}
```

Save this file and making sure to restart `$ ionic serve` in order for these changes to take effect.

Returning back to the *weather-service.ts* file, we can update the http.get to be:

```
this.http.get('/api/forecast/43.0742365,-89.381011899')...
```

Now, the ionic serve will now properly call the Dark Sky API and our live weather data will be returned to our app.

Connecting the Geolocation and Weather Providers

Currently, our latitude and longitude values are hard coded into the Dark Sky request. Let's address this issue.

Obviously, we need to know our current position. We have that code already in place with the Geolocation function. Since we are using the Ionic Native wrapper to the Cordova plugin, this is already sent up as a Promise. One of the advantages of using Promises is the ability to chain them together, which is exactly what we need to do here. Once we have our location, then we can call Dark Sky and get our weather data.

To chain Promises together, you just continue on using the .then function.

```
Geolocation.getCurrentPosition().then(pos => {
  console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
  this.currentLoc.lat = pos.coords.latitude;
  this.currentLoc.lon = pos.coords.longitude;
  this.currentLoc.timestamp = pos.timestamp;
  return this.currentLoc;
}).then(currentLoc => {
  weatherData.getWeather(currentLoc).then(theResult => {
    this.theWeather = theResult;
    this.currentData = this.theWeather.currently;
    this.daily = this.theWeather.daily;
    loading.dismiss();
  });
});
```

Besides adding the .then, we do have to add the return this.currentLoc within the Geolocation Promise. By adding a return, it enables us to pass this data along the Promise chain.

Demo Code Warning

We are being a bit sloppy here and not accounting for any errors. With any network calls to a remote system, you should always expect failure and code for that case.

This is a great example of how using Promises can make work with asynchronous processes so much easier.

Another minor tweak to the app is to remove the duration value in our loading dialog. We will let it stay up until we finally get our weather data. Once we do, we can simply call loading.dismiss(); and remove the loading dialog.

Now we need to update our *weather-service.ts* to support dynamic locations.

First, import our custom location class:

```
import { CurrentLoc } from '../interfaces/current-loc'
```

Next, change the load function to accept our current location:

```
load(currentLoc:CurrentLoc) {
```

Then modify the http.get call to reference this information:

```
this.http.get('/api/forecast/'+currentLoc.lat+','+currentLoc.lon)
```

Finally, we also need to adjust the getWeather method as well:

```
getWeather(currentLoc:CurrentLoc) {
  return this.load(currentLoc).then(data => {
    return data;
  });
}
```

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';
import { CurrentLoc } from '../interfaces/current-loc'

@Injectable()
export class WeatherService {
  data: any = null;

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

  load(currentLoc:CurrentLoc) {
    if (this.data) {
      return Promise.resolve(this.data);
    }

    return new Promise(resolve => {
      this.http.get('/api/forecast/'+currentLoc.lat+','+currentLoc.lon)
        .map(res => res.json())
        .subscribe(data => {
          this.data = data;
          resolve(this.data);
        });
    });
```

```
    }

    getWeather(currentLoc:CurrentLoc) {
      this.data = null;
      return this.load(currentLoc).then(data => {
        return data;
      });
    }
  }
```

One last change is to remove the call to `this.weatherData.load()` in *app.component.ts file*. Save the files, and reload the application and live local weather information will be shown on the page.

Other locations' weather

It is nice to know the weather where you are, but you might also want to know the weather in other parts of the world. We will use the sidemenu to switch to different cities.

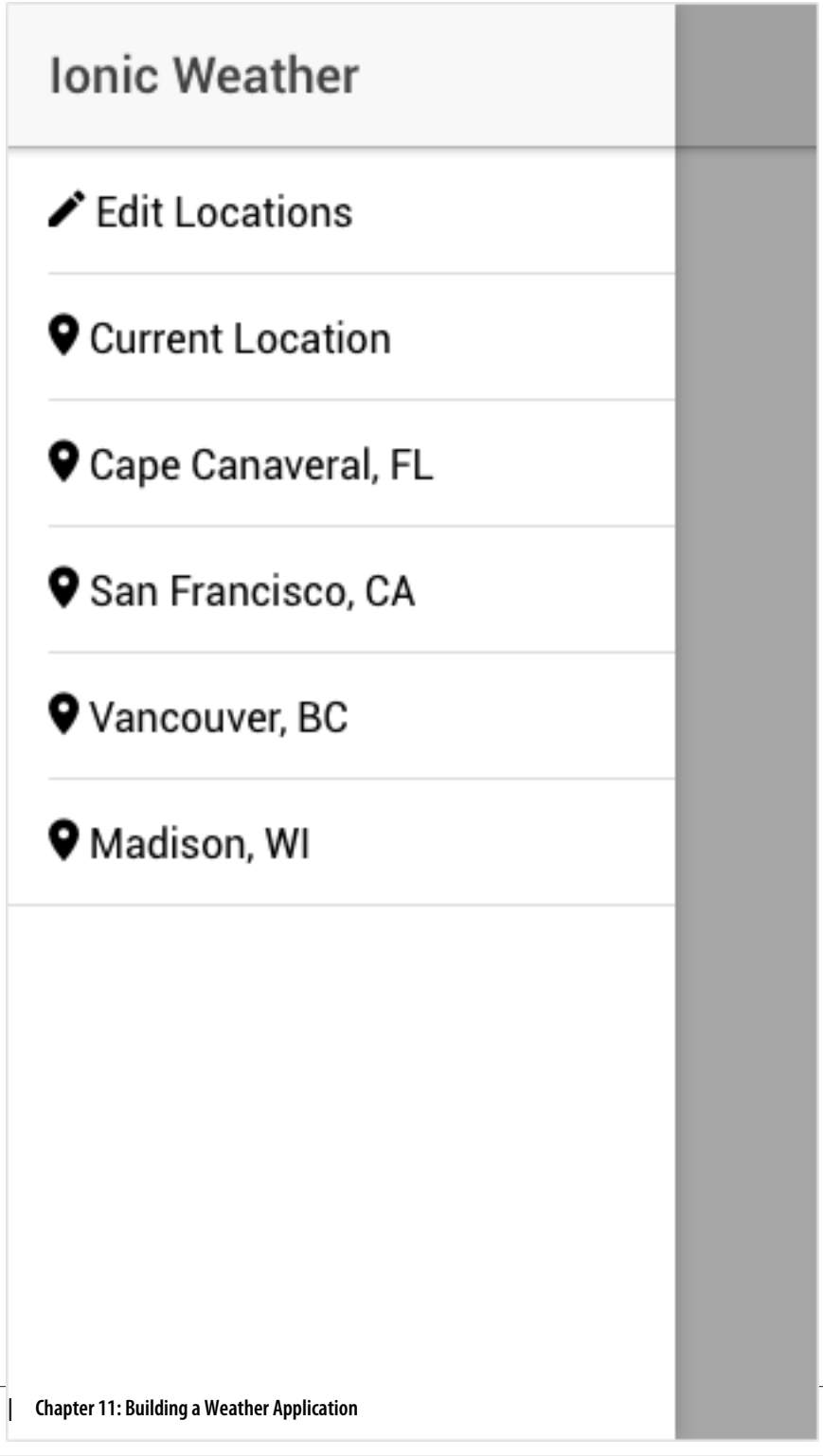


Figure 11-4. IonicWeather's sidemenu

The first set of changes we are going to make is to the pages array in the *app.component.ts* file. First, let's import our CurrentLoc class:

```
import { CurrentLoc } from '../interfaces/current-loc';
```

Next, we need to include the location's latitude and longitude data that we can reference to look up the weather. Since both the Edit Locations and Current Locations will not have predefined locations, this new property will be declared as optional.

```
pages: Array<{title: string, component: any, icon: string, loc?:CurrentLoc}>;
```

Next, we will fill this array with some locations:

```
this.pages = [
  { title: 'Edit Locations', component: Locations, icon: 'create' },
  { title: 'Current Location', component: Weather, icon: 'pin' },
  { title: 'Cape Canaveral, FL', component: Weather, icon: 'pin', ↵
    loc: {lat:28.3922, lon:-80.6077} },
  { title: 'San Francisco, CA', component: Weather, icon: 'pin', ↵
    loc: {lat:37.7749, lon:-122.4194} },
  { title: 'Vancouver, BC', component: Weather, icon: 'pin', ↵
    loc: {lat:49.2827, lon:-123.1207} },
  { title: 'Madison, WI', component: Weather, icon: 'pin', ↵
    loc: {lat:43.0742365, lon:-89.381011899} }
];
```

Now, when we select one of the four cities, we now know its latitude and longitude. But we need to pass this data along to our weather page. In Ionic 1, we would have used **$stateParams** to do this, but in Ionic 2 we can just pass an object as a parameter to the NavController. So, our openPage function will now pass this data (assuming it is there):

```
openPage(page) {
  // Reset the content nav to have just this page
  // we wouldn't want the back button to show in this scenario
  if (page.hasOwnProperty('loc') ) {
    this.nav.setRoot(page.component, {geoloc: page.loc});
  } else {
    this.nav.setRoot(page.component);
  }
}
```

Go ahead and save this file, and open *weather.ts*. Before we can access the data that is passed into this page, we need to import the NavParams module.

```
import { NavController, LoadingController, Refresher, NavParams } from 'ionic-angular';
```

Next, we need to update our constructor as well:

```
constructor(public navCtrl: NavController,
            public weatherData: WeatherService,
            public loadingCtrl: LoadingController,
            public navParams: NavParams) {
```

Retrieving the data is actually just a simple `get()` call. We will assign the data to a variable named **loc**. Add this code after we display the loading dialog:

```
let loc = this.navParams.get('geoloc');
```

If this variable is undefined, we will make the call to the Geolocation method like before. But, if this value is defined, then we will use that data to call the Weather service with it.

```
if (loc === undefined) {
  Geolocation.getCurrentPosition().then(pos => {
    console.log('lat: ' + pos.coords.latitude + ', lon: ' + pos.coords.longitude);
    this.currentLoc.lat = pos.coords.latitude;
    this.currentLoc.lon = pos.coords.longitude;
    this.currentLoc.timestamp = pos.timestamp;
    return this.currentLoc;
  }).then(currentLoc => {
    weatherData.getWeather(currentLoc).then(theResult => {
      this.theWeather = theResult;
      this.currentData = this.theWeather.currently;
      this.daily = this.theWeather.daily;
      loader.dismiss();
    });
  });
} else {
  this.currentLoc = loc;

  weatherData.getWeather(this.currentLoc).then(theResult => {
    this.theWeather = theResult;
    this.currentData = this.theWeather.currently;
    this.daily = this.theWeather.daily;
    loader.dismiss();
  });
}
```

For the case that we have a location, we will first assign it to the currentLoc variable, then call our getWeather function.

Save this file and test it out. Hopefully, the weather is different in each of the locations.

However, our page title is not updating after we switch locations. This is actually another simple fix. First, we need to pass the location's name along with its location. In *app.ts*, change

```
this.nav.setRoot(page.component, {geoloc: page.loc});
```

to

```
this.nav.setRoot(page.component, {geoloc: page.loc, title: page.title});
```

In *weather.ts*, we need to make two changes. First, we need a variable to store our page title in so the template can reference it. In the class constructor, we will add this.

```
pageTitle:string = 'Current Location';
```

Then, within our code block that gets the weather for the other locations, we can get the other NavParam and assign to the pageTitle.

```
this.pageTitle = this.navParams.get('title');
```

The last minor change is to the template itself. We need to update the `<ion-title>` tag to reference the pageTitle variable.

```
<ion-title>{{pageTitle}}</ion-title>
```

And with that, our page title should now display our current location's name.

Pull to refresh, part 2

You might have been wondering why we bothered to set the currentLoc to the location that was passed in? We can use it directly into our getWeather function. Remember that pull to refresh component we added? Yeah, that one. We actually never had it do anything except close after two seconds. Let's update this code to actually get new weather. All we have to do is replace the **setTimeout**, so the **doRefresh** function becomes:

```
doRefresh(refresher) {
  this.weatherData.getWeather(this.currentLoc).then(theResult => {
    this.theWeather = theResult;
    this.currentData = this.theWeather.currently;
    this.daily = this.theWeather.daily;
    refresher.complete();
  });
}
```

API Limits

Within the doRefresh would be a great place to check that timestamp property to prevent API abuse.

Unfortunately, I doubt you will see any changes to the weather data, as weather usually does not change that quickly. Let's turn our attention to the Edit Location screen.

Editing the Locations

This screen will be where we can add a new city to our list, or remove an existing one. Here is what our final screen will look like:

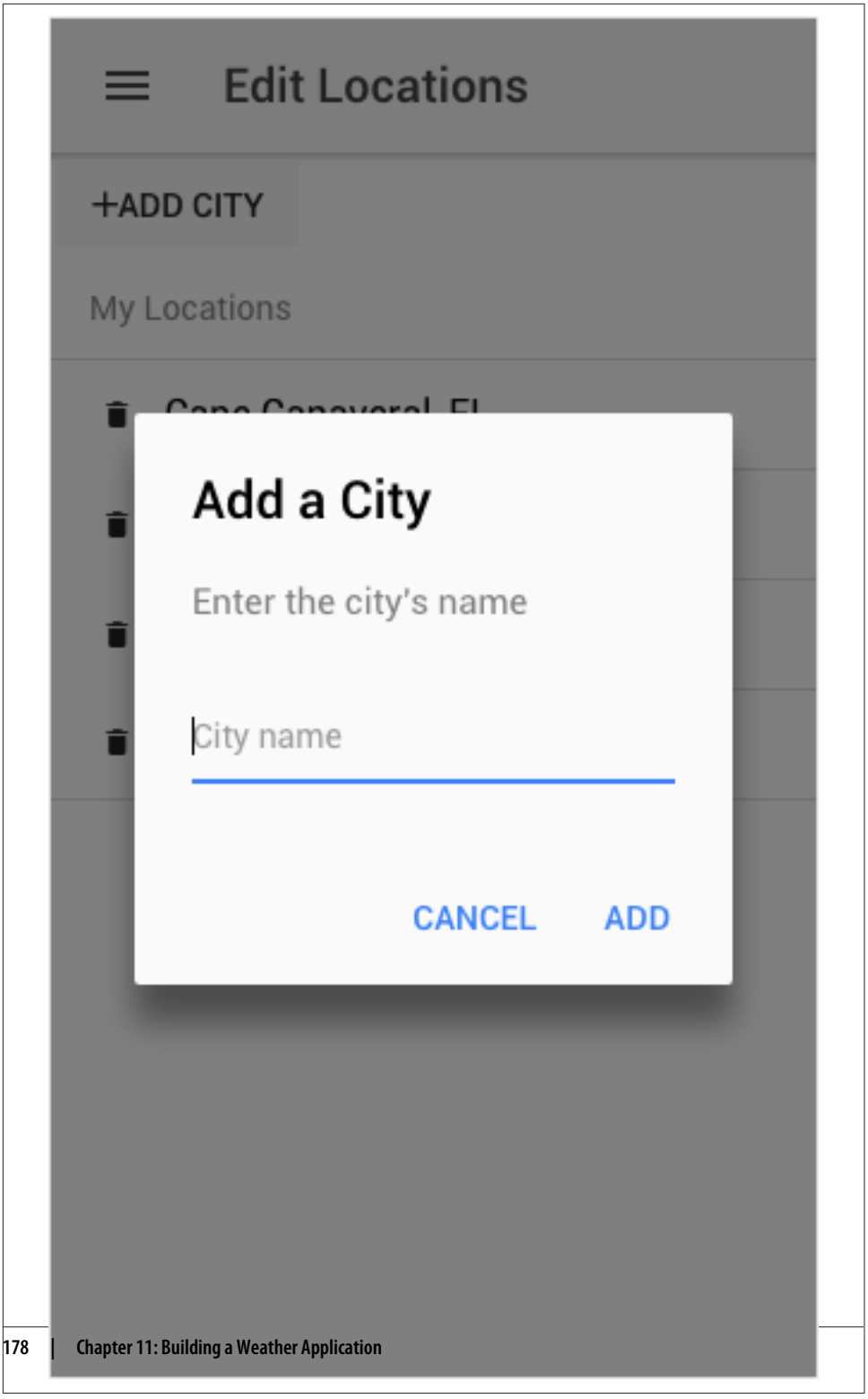


Figure 11-5. The Add City dialog

We will start with the <ion-content> let's add in our add city button.

```
<ion-content>
  <button ion-button icon-left clear color="dark" item-left (click)="addLocation()">↵
    <ion-icon name="add"></ion-icon>Add City</button>
</ion-content>
```

For this button, we are applying the clear and dark styles to it. The clear attribute will remove any border or background from the button, while the dark attribute will color the icon with the dark color. We will have the click event call a function named addLocation.

After the <button> tag, we are going to add in our <ion-list> that will display the list of our saved locations.

```
<ion-list>

  <ion-list-header>
    My Locations
  </ion-list-header>

  <ion-item *ngFor="let loc of locs">
   <button ion-button icon-left clear color="dark" (click)="deleteLocation(loc)">
     <ion-icon name="trash"></ion-icon>
   </button>{{loc.title}}
  </ion-item>

</ion-list>
```

We are using the <ion-list-header> to act as our list header rather than just a regular header tag. Hopefully, the <ion-item> tag looks a bit familiar to you. It is just going to iterate over an array of locs and render out each location. Next, we set up a button that will allow us to delete a location. Again, we will use the clear and dark attributes for the button styling. The click event will call a function named deleteLocation that we have yet to write. We will wrap up the code block by binding the text to the title value with {{loc.title}}.

With the HTML in place, we can focus on the changes we need to make in our code. We will start with *locations.ts* and add in the elements needed to get the template to work at a basic level.

We will again import our CurrentLoc class:

```
import { CurrentLoc } from '../../interfaces/current-loc';
```

Within the class definition, we need to create our locs array. This array is actually going to be the same content as our pages array does in the *app.component.ts* file.

```
export class Locations {
  locs: Array<{ title: string, component: any, icon: string, loc?: CurrentLoc }>;
```

But whenever you find yourself repeating elements like this, you should always pause and consider if this is something that should be abstracted. In this case, our answer is going to be yes. Instead of listing out our array's structure, we will move into to an interface and simply use it instead.

Create a new file named *weather-location.ts* inside the interfaces directory. Here we will define our WeatherLocation interface. Since the loc property uses the CurrentLoc interface, we need to make sure we import that module as well.

```
import { CurrentLoc } from './current-loc';

export interface WeatherLocation {
  title: string;
  component: any;
  icon: string;
  loc?: CurrentLoc;
}
```

The actual interface will define the location's title, it's page component, it's icon and an optional element of a CurrentLoc type.

With this interface created, switch back to *locations.ts* and import that interface:

```
import { WeatherLocation } from '../../interfaces/weather-location';
```

Then, we can update the locs array to be a properly typed array.

```
locs: Array<WeatherLocation>;
```

Next, let's add the two placeholder functions for the adding and deleting of locations after the constructor.

```
deleteLocation(loc) {
  console.log('deleteLocation');
}

addLocation() {
  console.log('addLocation');
}
```

Let's work on populating our locs array with our saved places. We could simply copy over the pages array that is used in the sidemenu and our template should render it out just fine. But any changes we make to our locs array will not be reflected in the pages array. We need to move this data into something that both components can reference.

For this, we will create a new provider named **LocationsService**. Instead of using the ionic generate command, which creates a provider that is aimed at getting remote data, will we just manually create our provider.

Create a new file named *locations-service.ts* within the providers directory.

The first thing we need to do is define our imports. We will need Injectable from Angular, as well as our WeatherLocation interface and WeatherPage component.

```
import { Injectable } from '@angular/core';
import { WeatherLocation } from '../interfaces/weather-location';
import { Weather } from '../pages/weather/weather';
```

Next, we need to include the @Injectable() decorator.

```
@Injectable()
```

Now, we can define our service, the locations Array (properly typed to WeatherLocation), and initialize that array with our default locations:

```
export class LocationsService {
  locations: Array<WeatherLocation>;

  constructor() {
    this.locations = [
      { title: 'Cape Canaveral, FL', component: Weather, icon: 'pin', ↵
        loc: { lat: 28.3922, lon: -80.6077 } },
      { title: 'San Francisco, CA', component: Weather, icon: 'pin', ↵
        loc: { lat: 37.7749, lon: -122.4194 } },
      { title: 'Vancouver, BC', component: Weather, icon: 'pin', ↵
        loc: { lat: 49.2827, lon: -123.1207 } },
      { title: 'Madison, WI', component: Weather, icon: 'pin', ↵
        loc: { lat: 43.0742365, lon: -89.381011899 } }
    ];
  }
```

We can wrap our service with the functions that we will need, one to get the locations, one to add to the locations, and one function to remove a location.

```
getLocations() {
    return Promise.resolve(this.locations);
}

removeLocation(loc) {
    let index = this.locations.indexOf(loc)
    if (index != -1) {
        this.locations.splice(index, 1);
    }
}

addLocation(loc) {
    this.locations.push(loc);
}
```

As with all Providers, we need to make sure they are included in our provider array in *app.module.ts*.

```
providers: [WeatherService, GeocodeService, LocationsService]
```

Returning back to *locations.ts*, we can import this service into the component.

```
import { LocationsService }  from '../../providers/locations-service';
```

We also need to add this module into our constructor, and then call that service's **getLocations** method to get our default locations and save the result into our locs array.

```
constructor(public navCtrl: NavController,
            public locations: LocationsService) {
  locations.getLocations().then(res => {
    this.locs = res;
  });
}
```

Saving the file, and navigating to the Edit Locations screen, you should now see our four default cities listed.

Deleting a City

Let's implement the actual **deleteLocation** function in the *locations.ts* file. We already have the stub function in place in our component, all we need to do is simply call the removeLocation method on the LocationsService.

```
deleteLocation(loc) {
  this.locations.removeLocation(loc);
}
```

Adding a City

Adding a city is a bit more complex. If you recall, the Dark Sky weather service uses latitude and longitudes to look up the weather data. But I doubt you will ask "How is the weather in 32.715, -117.1625 ?", but rather "How is the weather in San Diego?". To translate between a location and its corresponding latitude and longitude we need to use a geocoding service.

Using a Geocode service

For this app, we will use the Geocoding API from Google. You can learn more about the API at https://developers.google.com/maps/documentation/geocoding/intro

In order to use this API, we will need to be registered with Google as a developer and have an API key for our application. Go to https://developers.google.com/maps/documentation/geocoding/get-api-key and follow the instructions to generate an API key for the Geocoding API. Save this 40 character string as we will need it shortly.

```
export class GeocodeService {
  data: any;
  apikey:String = 'YOUR-API-KEY-HERE';
  constructor(public http: Http) {
    this.data = null;
  }

  getLatLong(address:string) {
    if (this.data) {
```

```
      // already loaded data
      return Promise.resolve(this.data);
    }

    // don't have the data yet
    return new Promise(resolve => {
      this.http.get('https://maps.googleapis.com/maps/api/geocode/↵
      json?address='+encodeURIComponent(address)+'&key='+this.apikey)
        .map(res => res.json())
        .subscribe(data => {
            if(data.status === "OK") {
                resolve({name: data.results[0].formatted_address, location:{
                    latitude: data.results[0].geometry.location.lat,
                    longitude: data.results[0].geometry.location.lng
                }});
            } else {
                console.log(data);
                //reject
            }
        });
    });
  }
}
```

The first change is the new variable named **apikey**. You will need to set its value to the key you just generated.

The constructor just sets the data variable to null. The heart of this class is actually the getLatLong method. This method accepts one parameter named address.

The method will then make this request:

```
this.http.get('https://maps.googleapis.com/maps/api/geocode/json?↵
address='+encodeURIComponent(address)+'&key='+this.apikey)
```

Note that we have to use the **encodeURIComponent** to sanitize the address string before we can make the call to the Google Geocoding API.

Once the data is returned from the service, we can check if a named place was located using the data.status value. Then we can traverse the JSON and get the needed data and resolve the Promise. Here is the completed method:

```
getLatLong(address:string) {
  if (this.data) {
    // already loaded data
    return Promise.resolve(this.data);
  }

  // don't have the data yet
  return new Promise(resolve => {
    this.http.get('https://maps.googleapis.com/maps/api/geocode/↵
    json?address='+encodeURIComponent(address)+'&key='+this.apikey)
      .map(res => res.json())
```

```
        .subscribe(data => {
            if(data.status === "OK") {
                resolve({name: data.results[0].formatted_address, location:{
                    latitude: data.results[0].geometry.location.lat,
                    longitude: data.results[0].geometry.location.lng
                }});
            } else {
                console.log(data);
                //reject
            }
        });
    });
}
```

Now that we have a way to turn San Diego into 32.715, -117.1625. We can return back to *locations.ts* and finish our addLocation function.

We will need to import our new Geocode Service.

```
import { GeocodeService } from '../../providers/geocode-service';
```

and also include it with the constructor.

```
 constructor(public navCtrl: NavController,
             public locations: LocationsService,
             public geocode: GeocodeService) {
```

Instead of using the Ionic Native Dialog plugin to prompt our user to enter their point of interest, let's use the AlertController component. Again, we need to import it from the proper library.

```
import { NavController, AlertController } from 'ionic-angular';
```

One of the difficulties in using the Ionic Native Alert is the degree to which you can extend it, as well as requiring you to test your application either in a simulator or on-device. By using the standard Ionic Alert component, we can keep developing directly in our browser. However, unlike the Ionic Native dialog, there is quite a bit more Javascript that you need to write. We will need to update our constructor to include the AlertController:

```
constructor(public navCtrl: NavController,
              public locations: LocationsService,
              public geocode: GeocodeService,
              public alertCtrl: AlertController) {
```

Here is the completed **addLocation** method with the Alert component used.

```
addLocation() {
  let prompt = this.alertCtrl.create({
    title: 'Add a City',
    message: "Enter the city's name",
    inputs: [
      {
        name: 'title',
```

```
          placeholder: 'City name'
        },
      ],
      buttons: [
        {
          text: 'Cancel',
          handler: data => {
            console.log('Cancel clicked');
          }
        },
        {
          text: 'Add',
          handler: data => {
            console.log('Saved clicked');
          }
        }
      ]
    });

    prompt.present();
  }
```

The key line of code not to forget is telling the AlertController to present our alert.

Now, we are not doing anything with any city that you might enter, so let's add that code. Replace this line

```
console.log('Saved clicked');
```

with

```
if (data.title != '') {
  this.geocode.getLatLong(data.title).then(res => {
    let newLoc = { title: '', component: Weather, icon: 'pin', ↵
                   loc: { lat: 0, lon: 0 } }
    newLoc.title = res.name;
    newLoc.loc.lat = res.location.latitude;
    newLoc.loc.lon = res.location.longitude;

    this.locations.addLocation(newLoc);
  });
}
```

We will also need to import our Weather component as well:

```
import { Weather } from '../weather/weather';
```

So now we can take the inputted city name, parse it into a latitude and longitude value and add it to our list of locations.

There are a couple of open issues I do want to point out with this code block. The first issue is that we are not handling the case of when the Geocoding service fails to find a location. If you are up for that programming challenge you need to add code

that rejects the Promise. The second issue is handling when the Geocoding service returns more than one answer. For example, if I enter Paris, do I want Paris, France or Paris, Texas? For this solution, you might want to look at the flexibility of the Alert component to have radio buttons in a dialog.

Dynamically Updating the Sidemenu

If add a new location to your list, then use the sidemenu to view that location's weather, you will see that it is not listed. That is because that list is still referencing the local version and not using the array in LocationsServices. We will open the *app.component.ts* file and refactor it.

First, we need to import the service:

```
import { LocationsService } from '../../providers/locations-service';
```

Next, we need to include in our constructor:

```
constructor(
  public platform: Platform,
  public locations: LocationsService
) {
  this.initializeApp();
  this.getMyLocations();
}
```

The **getMyLocations** function will get the data from the LocationsService provider and then populate the pages array.

```
getMyLocations(){
  this.locations.getLocations().then(res => {
    this.pages = [
      { title: 'Edit Locations', component: Locations, icon: 'create' },
      { title: 'Current Location', component: Weather, icon: 'pin' }
    ];
    for (let newLoc of res) {
      this.pages.push(newLoc);
    }
  });
}
```

Save all the files and run `$ ionic serve` again. The sidemenu should still show our initial list of places. If we add a new location, the Edit Locations screen is properly updated. However, the sidemenu is not showing our new location. Even though the sidemenu is getting its data from our shared Provider. It does not know that the data has changed. To solve this issue, we need to explore two different options: Ionic Events and Observables.

Ionic Events

The first solution is to use Ionic Events to communicate the change in the dataset. According to the documentation

> Events is a publish-subscribe style event system for sending and responding to application-level events across your app.

Sounds pretty close to what we need to do. In the *locations.ts* file, we will need to import the Events component from ionic-angular.

```
import { NavController, Alert, Events } from 'ionic-angular';
```

Within the constructor we need to have a reference to the events module:

```
constructor(
  private nav: NavController,
  public geocode: GeocodeService,
  public locations: LocationsService,
  public events: Events)
```

Now, in both the **deleteLocation** and **addLocation** functions, we just need to add this code to publish the event.

```
this.events.publish('locations:updated', {});
```

So the **deleteLocation** becomes:

```
deleteLocation(loc) {
  this.locations.removeLocation(loc);
  this.events.publish('locations:updated', {});
}
```

For the addLocation function, it is added within the 'Add' handler callback:

```
handler: data => {
  if (data.title != '') {
    this.geocode.getLatLong(data.title).then(res => {
      let newLoc = { title: '', component: Weather, icon: 'pin', ↵
                     loc: { lat: 0, lon: 0 } }
      newLoc.title = res.name;
      newLoc.loc.lat = res.location.latitude;
      newLoc.loc.lon = res.location.longitude;

      this.locations.addLocation(newLoc);
      this.events.publish('locations:updated', {});
    });
  }
}
```

The parameters are the event name, in this case, locations:updated, and any data that needs to be shared. For this example, there is no additional data we need to send to the subscriber.

The subscriber function will be added to the *app.component.ts* file. First, we need to update our imports:

```
import { Nav, Platform, Events } from 'ionic-angular';
```

and pass it into our constructor:

```
constructor(public platform: Platform,
            public locations: LocationsService,
            public events: Events) {
```

Within the constructor itself, we will add this code:

```
this.initializeApp();
this.getMyLocations();
events.subscribe('locations:updated', (data) => {
  this.getMyLocations();
});
```

This code will listen for a '**locations:updated**' event, then it will call the **getMyLocations** function. In doing so, our array will be refreshed, and the sidemenu will be kept up to date.

Using Ionic Events may not be the best solution to this problem, but it is worth knowing about how to communicate events across an application.

Observables

You might be wondering if there was some other way for the data updates to propagate through our app without the need to manually send events. In fact, there is a solution available to us. One of the elements inside the RxJS library is Observables. From their documentation:

> The Observer and Observable interfaces provide a generalized mechanism for push-based notification, also known as the observer design pattern. The Observable object represents the object that sends notifications (the provider); the Observer object represents the class that receives them (the observer).

Meaning, the event notification system that we wrote with Ionic Events can be replaced. In our simple example using Ionic Events, we did not have to write a lot of code for our system to work. Now imagine a much more complex app and the messaging infrastructure that could quickly become a gordian knot.

Now, to say that RxJS is a powerful and complex library is an understatement. But here are the basics of what we need to create in our app. First, we need to create a RxJS Subject, specifically a Behavior Subject. This will hold our data that we wish to monitor for changes. Next, we need to create the actual Observable that will watch our subject for changes. If it sees a change, it will broadcast the new set of data. The third and final part are the subscribers to our Observable. Once they are bound together, our data will always be the latest.

We will start our refactoring in the *locations-service.ts* file. This is where the majority of the changes will occur as change our app from using Ionic Events to Observables.

As always, we need to import the needed modules:

```
import { Observable, BehaviorSubject } from 'rxjs/Rx';
```

Now, there are several types of Subject in the RxJS library. The BehaviorSubject is the best type of our needs, as it will send an update as soon as it first gets data. Other types of Subjects require an additional method call to broadcast an update.

Next, we need to define the actual BehaviorSubject and Observable within the class

```
locations: Array<WeatherLocation>;
locationsSubject: BehaviorSubject<Array<WeatherLocation>> = new BehaviorSubject([]);
locations$: Observable<Array<WeatherLocation>> = this.locationsSubject.asObservable();
```

Variable Naming

The inclusion of the **$** after the variable name is a naming convention for Observable data types. To learn more about Angular best practices, see John Papa's Style Guide at https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md

We have only associated the **locations$** with the locationsSubject by setting it to **this.locationsSubject.asObservable()**. The locationsSubject knows nothing about our data in the locations array. To solve this issue, simply pass in our locations array as the parameter to the locationsSubject's next method.

```
this.locationsSubject.next(this.locations);
```

Upon doing this, our Observable will emit an update event, and all the references will be change. Rather than repeating this call whenever our locations array changes, let's wrap it in a refresh function:

```
refresh() {
    this.locationsSubject.next(this.locations);
}
```

Then in the **constructor**, **addLocation**, and **removeLocation** methods, we can call this method after we have done whatever changes we needed to make to the locations array. Here is the completed code:

```
import { Injectable } from '@angular/core';
import { WeatherLocation } from '../interfaces/weather-location';
import { Weather } from '../pages/weather/weather';
import { Observable, BehaviorSubject } from 'rxjs/Rx';

@Injectable()

export class LocationsService {
  locations: Array<WeatherLocation>;
  locationsSubject: BehaviorSubject<Array<WeatherLocation>> = new BehaviorSubject([]);
  locations$: Observable<Array<WeatherLocation>> = this.locationsSubject.asObservable();

  constructor() {
    this.locations = [
      { title: 'Cape Canaveral, FL', component: Weather, icon: 'pin', ↵
```

```
        loc: { lat: 28.3922, lon: -80.6077 } },
      { title: 'San Francisco, CA', component: Weather, icon: 'pin', ↵
        loc: { lat: 37.7749, lon: -122.4194 } },
      { title: 'Vancouver, BC', component: Weather, icon: 'pin', ↵
        loc: { lat: 49.2827, lon: -123.1207 } },
      { title: 'Madison, WI', component: Weather, icon: 'pin', ↵
        loc: { lat: 43.0742365, lon: -89.381011899 } }
    ];
    this.refresh();
  }

  getLocations() {
    return Promise.resolve(this.locations);
  }

  removeLocation(loc) {
    let index = this.locations.indexOf(loc)
    if (index != -1) {
      this.locations.splice(index, 1);
      this.refresh();
    }
  }

  addLocation(loc) {
    this.locations.push(loc);
    this.refresh();
  }

  refresh() {
    this.locationsSubject.next(this.locations);
  }
}
```

With the service converted to using an Observable, we need to create the data subscribers to it. In *locations.ts* we now need to replace

```
locations.getLocations().then(res => {
  this.locs = res;
});
```

with

```
locations.locations$.subscribe( ( locs: Array<WeatherLocation> ) => {
  this.locs = locs;
});
```

Now our **locs** array will be automatically updated whenever the data changes in our service. Go ahead and remove the two **this.events.publish** calls.

A similar change is needed the *app.component.ts* file. First, let's remove the event listener.

```
events.subscribe('locations:updated', (data) => {
  this.getMyLocations();
});
```

Next, we need to include the WeatherLocation interface in the imports

```
import { WeatherLocation } from '../interfaces/weather-location';
```

Finally, we can replace the getMyLocations function

```
getMyLocations(){
  this.locations.getLocations().then(res => {
    this.pages = [
      { title: 'Edit Locations', component: LocationsPage, icon: 'create' },
      { title: 'Current Location', component: WeatherPage, icon: 'pin' }
    ];
    for (let newLoc of res) {
      this.pages.push(newLoc);
    }
  });
}
```

with

```
getMyLocations(){
  this.locations.locations$.subscribe( ( locs: Array<WeatherLocation> ) => {
      this.pages = [
        { title: 'Edit Locations', component: Locations, icon: 'create' },
        { title: 'Current Location', component: Weather, icon: 'pin' }
      ];
      for (let newLoc of locs) {
        this.pages.push(newLoc);
      }
    } );
}
```

With that, our weather app is now using RxJS Observables. Observables are a power solution when working with dynamic data. You would be well served to spend some time exploring their capabilities. Let's now turn our attention to making our app a bit visually pleasing.

Styling the app

With our app now functioning rather well, we can turn our attention to some visual styling. Included with the source code is a nice photograph of a partially cloudy sky. Let's use this as our background image. Since our Ionic apps are based on HTML and CSS, we can leverage our existing CSS skills to style our apps. The only challenge in working with Ionic is uncovering the actual HTML structure that our CSS needs to properly target.

With the improved selector system within Ionic, targeting a specific HTML element is much easier. Let's include a nice sky image to serve as our background for the Weather page. Open the *weather.scss* and add the following CSS:

```
page-weather {
  ion-content{
    background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
    -webkit-background-size: cover;
    background-size: cover;
  }
}
```

Writing Sass

The CSS that we are adding should be nested within the page-weather {}.

Now, let's increase our base font size as well to make the text more readable.

```
ion-content{
  background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
  -webkit-background-size: cover;
  background-size: cover;
  font-size: 24px;
}
```

However, the default black text is not really working against the sky and clouds. So we can adjust the `ion-col` and its two children. We will change the color to white, center the text, and apply a drop shadow that has a little transparency.

```
ion-col, ion-col h1, ion-col p {
    color: #fff;
    text-align: center;
    text-shadow: 3px 3px 3px rgba(0,0, 0, 0.4);
}
```

Now, adjust the current weather information's text:

```
h1 {
    font-size: 72px;
}

p {
    font-size: 36px;
    margin-top: 0;
}
```

But, what about the header? It is looking a bit drab and out of place. Let's add a new class, **opaque**, to the **`<ion-header>`** tag in the *weather.html* file.

In the *weather.scss* file we can apply a series of classes to give our header a more modern look. The first class we will add will use the new backdrop-filter method.

```
.opaque {
  -webkit-backdrop-filter: saturate(180%) blur(20px);
  backdrop-filter: saturate(180%) blur(20px);
}
```

Unfortunately, backdrop-filter is only supported in Safari at this time. This means we need to also create a fallback solution.

```
.opaque .toolbar-background {
  background-color: rgba(#f8f8f8, 0.55);
}
```

With this pair of CSS classes, we have much more current design style in place.

But there are still more elements to style on this screen, namely the Refresher component. This component is a bit more complex that components we have styled before. Namely, it has a series of states that each need to be styled.

Let's start with the two text elements; the pulling text and the refreshing text. Unfortunately, the current documentation does not list out the structure or style method. But with a little inspection with the Chrome Dev Tools, these elements did already have CSS classes applied to them. So we can just add the following:

```
.refresher-pulling-text, .refresher-refreshing-text  {
    color:#fff;
}
```

The arrow icon is also easily styled using

```
.refresher-pulling-icon {
    color:#fff;
}
```

But what about the spinner? The dark circles aren't really standing out against our background. This element is bit tricky to style as well. The spinner is actually a SVG element. Which means we can not change it just by changing the CSS of the `<ion-spinner>`, but instead we need to modify the values within the SVG. One thing to note is that some of the CSS properties on an SVG element have different names. For example, SVG uses the term stroke instead of border, and fill instead of background-color.

If you are using the circles option for your refreshingSpinner attribute, then the styling is:

```
.spinner-circles circle{
    fill:#fff;
}
```

But, say you decided to use crescent as your spinner, then the styling would be:

```
.spinner-crescent circle{
    stroke: #fff;
}
```

There is one last item that we should change. You probably did not see that while the refresher is visible, there was a solid 1 pixel white line between the refresher and the content. With our full background image, this doesn't quite work for this design. Again, with some inspection with Chrome Dev Tools, the CSS was found where this attribute was set. So, we can now override it with:

```
.has-refresher > .scroll-content {
    border-width: 0;
}
```

Here is the full *weather.scss* code:

```
page-weather {
  ion-content{
    background: url(../assets/imgs/bg.jpg) no-repeat center center fixed;
    -webkit-background-size: cover;
    background-size: cover;
    font-size: 24px;
  }

  ion-col, ion-col h1, ion-col p {
    color: #fff;
    text-align: center;
    text-shadow: 3px 3px 3px rgba(0,0, 0, 0.4);
  }

  h1 {
    font-size: 72px;
  }

  p {
    font-size: 36px;
    margin-top: 0;
  }

  .opaque {
    -webkit-backdrop-filter: saturate(180%) blur(20px);
    backdrop-filter: saturate(180%) blur(20px);
  }

  .opaque .toolbar-background {
    background-color: rgba(#f8f8f8, 0.55);
  }

  .refresher-pulling-text, .refresher-refreshing-text  {
    color:#fff;
  }

  .refresher-pulling-icon {
```

```
    color:#fff;
  }

  .spinner-circles circle{
    fill:#fff;
  }

  .has-refresher > .scroll-content {
    border-top-width: 0;
  }
}
```

Figure 11-6. Our styled IonicWeather app

With that our main weather page is looking rather nice. But we can add one more touch to the design. How about a nice icon as well?

Add a weather icon

The Dark Sky data set actually defines an icon value, and the Ionicons also support several weather icons as wells. Here is a table mapping each icon.

Dark Sky name	Ionicon name
clear-day	sunny
clear-night	moon
rain	rainy
snow	snow
sleet	snow
wind	cloudy
fog	cloudy
cloudy	cloudy
partly-cloudy-day	partly-sunny
partly-cloudy-night	cloudy-night

We don't have a complete one to one mapping, but it is close enough. To resolve the mapping between the Dark Sky name and the Ionicon name, we will create a custom Pipe function. If you recall, Pipes are functions that transform data in a template.

From our terminal, we can use the Ionic generate command to scaffold our pipe for us:

```
$ ionic generate pipe weathericon
```

This will create a new directory named pipes, and a new file named *weathericon.ts*. Here is the stock code that Ionic will generate for us:

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

/*
  Generated class for the Weathericon pipe.

  See https://angular.io/docs/ts/latest/guide/pipes.html for more info on
  Angular 2 Pipes.
*/
@Pipe({
  name: 'weathericon'
})
@Injectable()
export class Weathericon {
  /*
    Takes a value and makes it lowercase.
```

```
   */
  transform(value, args) {
    value = value + ''; // make sure it's a string
    return value.toLowerCase();
  }
}
```

Let's replace the code within the **transform** function. The goal of this function will be to take the Dark Sky icon string and find the corresponding Ionicon name. Here is the revised Pipe:

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

@Pipe({
  name: 'weathericon'
})
@Injectable()
export class Weathericon {

  transform(value: string, args: any[]) {
    let newIcon:string ='sunny';
    let forecastNames:Array<string> = ["clear-day", "clear-night", "rain", ↵
"snow", "sleet", "wind", "fog", "cloudy", "partly-cloudy-day", "partly-cloudy-night"];
    let ioniconNames:Array<string> = ["sunny", "moon", "rainy", "snow",↵
"snow", "cloudy", "cloudy", "cloudy", "partly-sunny", "cloudy-night"];
    let iconIndex:number = forecastNames.indexOf(value);
    if (iconIndex !== -1) {
      newIcon = ioniconNames[iconIndex];
    }

    return newIcon;
  }
}
```

In our *app.module.ts* file, we will need to import our pipe using:

```
import { Weathericon } from '../pipes/weatherIcon';
```

and then include Weathericon in the declarations array.

Finally, the last change is to markup in the *weather.html* file which will become

```
<ion-col width-100>
  <h1> {{currentData.temperature | number:'.0-0'}}&deg;</h1>
  <p><ion-icon name="{{currentData.icon | weathericon}}"></ion-icon>↵
 {{currentData.summary}}</p>
</ion-col>
```

Figure 11-7. IonicWeather app with an weather Ionicon.

That really does finish off the look of this screen. Feel free to continue to explore styling the sidemenu and the locations page on your own.

Next Steps

Our weather app still has some things that can be improved upon. Most notably, our custom cities are not saved. For something as simple as a list of cities, using Firebase might be overkill. Some options you might consider would be either local storage or using the Cordova File plugin to read and write a simple data file. The choice is up to you.

Another challenge you might consider would be to introduce dynamic backgrounds. You could use a Flickr API to pull an image based on the location, or based on the weather type.

One more challenge you could take on is to support changing the temperature units.