Hybrid Mobile Apps with Ionic and AngularJS

Ionic IN ACTION

Jeremy Wilken

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Ionic in Action
Version 6

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thanks for purchasing the MEAP of *Ionic in Action: Hybrid Mobile Apps with Ionic and AngularJS*. I hope you find it useful. I'm looking forward to hearing from you in the Author Online forum—let me know how to make this book better!

The Ionic Framework is a combination of tools and utilities that enable developers to quickly build hybrid mobile apps using the same technologies used to build websites and web applications. With the Ionic Framework and AngularJS, you'll begin building hybrid mobile apps that look and feel just like a native mobile app.

By the end of the book you should understand:

- How to setup your development environment for Ionic
- How to use AngularJS as the foundation for your app
- How to use Ionic to build stunning mobile interfaces
- How to employ Ionic Services to further manage the UI
- How to use Cordova to integrate with device features, like GPS
- How to test and deploy your app to stores

The book is divided into three parts.

In Part I, you'll install your working and testing environment for either Android or iOS, get familiar with it, and build your first app. I'll also teach you some foundations of AngularJS that we'll be using in this book. I'll explain things clearly for you and have created several graphics to give you an idea of how everything fits together.

In Part II, you'll learn how to use Ionic to create and manipulate a user interface optimized for both users and devices. Your users will think they are using an app developed using the native SDK. You'll be able to build complex interfaces using Ionic's components, and see how they work together.

In Part III, you will learn some advanced Ionic and Angular development skills and how to test and deploy your app. I'll show you how to use Cordova to access the device's features such as the camera. We'll work through different ways to test your app to get it deployment ready and I'll also show you how to configure and deploy your app for both the iOS App Store and the Android Play Store.

Tell me what you think of what I've written and what you'd like to see in the rest of the book. Your feedback through the AO forum will be very valuable as I write and improve *Ionic in Action: Hybrid Mobile Apps with Ionic and AngularJS*.

Thanks again for your interest and for purchasing the MEAP!

—Jeremy Wilken

brief contents

1

Ionic and Hybrid Apps

This chapter covers

- Why to choose Ionic and how it benefits you.
- What Ionic is and how it uses AngularJS and Cordova.
- Why hybrid apps are an ideal choice for mobile development.
- Introduction and requirements for Android and iOS platforms

Congratulations, you've picked up the best resource to begin building hybrid mobile apps that look and feel just like a native mobile app using the Ionic Framework. The Ionic Framework is a combination of tools and utilities that enable developers to quickly build hybrid mobile apps using the same technologies used to build websites and web applications. Ionic works by embedding a web application inside of a native app by using Cordova. It is designed to work together with AngularJS to create a web application for the mobile environment, and includes support for mobile features like responding to touch inputs and user interface controls.

This book aims to give developers the skills necessary to build amazing hybrid apps using the same skills from building web applications. You might be a freelance developer who would like to offer clients the service of building custom mobile apps or a project manager in a Fortune 500 company trying to determine the viability of building a hybrid app for your company. There is a wide range of reasons why you might want to learn about building hybrid mobile apps. Before we get too far along, we should clarify what Ionic is and why it is a solid choice for building hybrid mobile apps.

1.1 What is Ionic?

The Ionic Framework is a combination of technologies and utilities designed to make building hybrid mobile apps fast, easy, and beautiful. Ionic doesn't run entirely by itself, it is built with AngularJS as the web application framework and is designed to work with Cordova for the

building of the native app. We'll dig into each in more detail later, but figure 1.1 shows an overview of the several technologies and how they stack. Let's take a moment to cover the basics of how the technology stack works on a device.

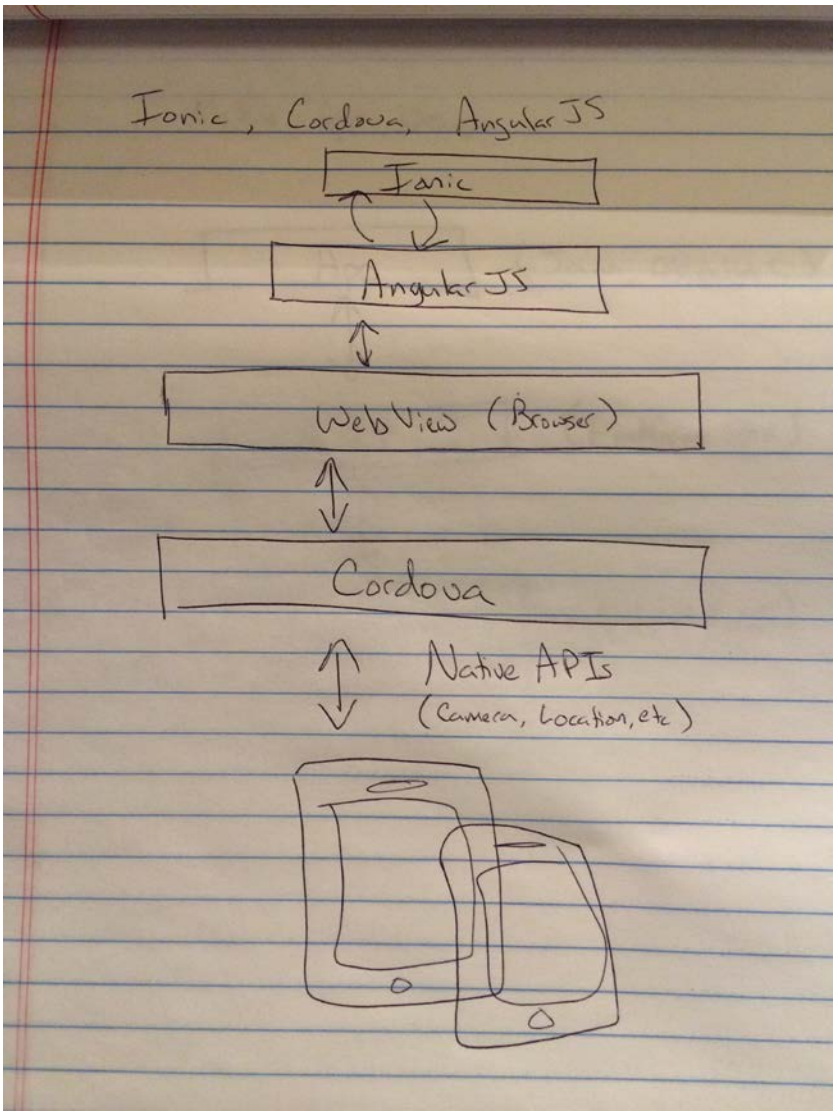


Figure 1.1 The stack of technologies used with the Ionic Framework, and how they fit together.

In figure 1.1, the stack begins with the device. Imagine this is an iPhone running iOS or a Nexus 10 running Android. The device contains the operating system that manages the installation of apps, which are downloaded from the platform's store. The operating system also provides a set of APIs for apps to use to access various features, like the GPS location, contacts list, or camera.

The next layer is Cordova, which is a utility for creating a bridge between the platform and our application. It creates a native mobile app that can be installed, and contains what is called a WebView (essentially an isolated browser window), which the web application will run inside. This generated native app is able to access both the web application and the native platform, since Cordova provides a JavaScript API for the web application to use to communicate with the native device. This is primarily handled behind the scenes, and Cordova ultimately generates the native app for you.

The last two technologies are AngularJS and Ionic. Inside of the WebView our AngularJS web application runs. AngularJS is primarily used to manage the web application's logic and data. Ionic is built on top of AngularJS, and is primarily used to design the user interface and experience. This includes the visual elements such as tabs, buttons, and navigation headers.

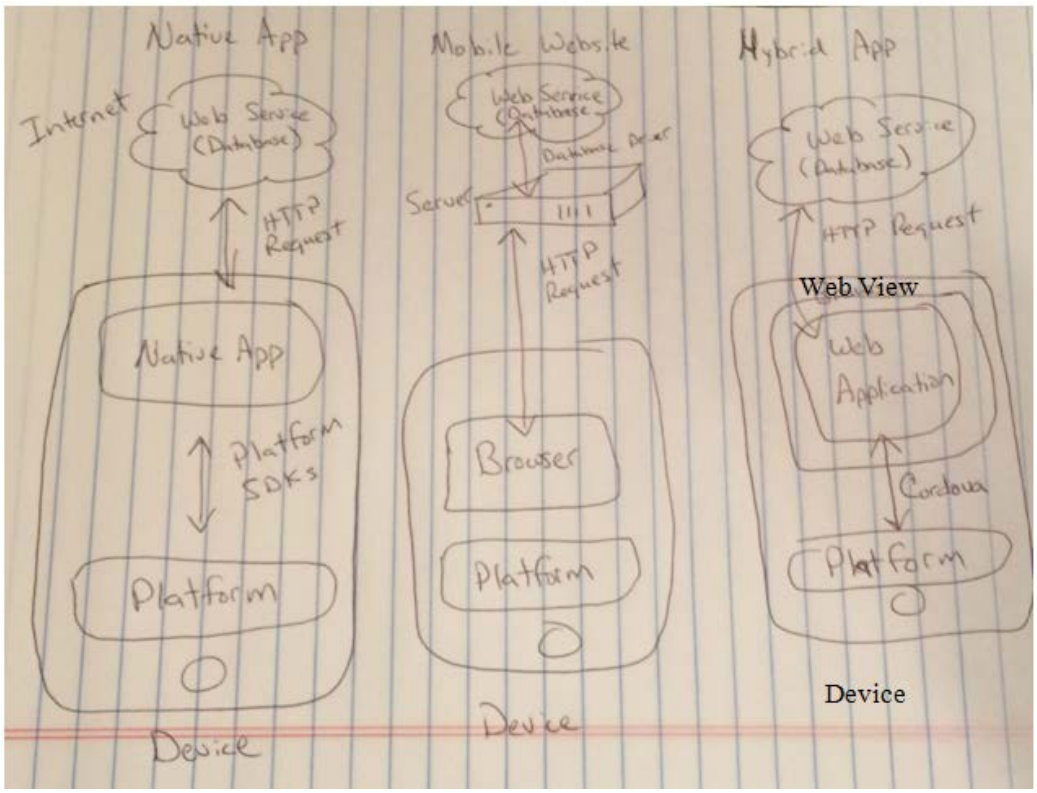
The core of Ionic is the user interface controls described above. However Ionic also includes a number of additional utilities and features that help manage your app from creation to previewing to deployment.

Now that you have a bird's eye view of Ionic and the technology, let's look a little closer at the types of mobile experiences and why Ionic's approach is beneficial.

1.2 Types of mobile experiences

It is important to understand there are several ways to build applications for mobile devices, and each has strengths and weaknesses. There are three basic types, native apps, mobile websites, and hybrid apps. We'll look at each of these in detail to clarify the differences.

In the following figure, you can see how the three types compare in design and architecture. It shows also how the apps would access a database or web service API to load data.



Native apps, mobile websites, and hybrid app architectures compared side by side.

1.2.1 Native Mobile Apps

Native apps are written using the default language for the mobile platform, which is Objective C for iOS (soon to be Swift for iOS8) or Java for Android. Native apps are compiled and execute directly on the device. Using the platform SDK (API), the app can communicate with the platform to access device data or load data from an external website using http requests.

Both iOS and Android provide a set of tools to enable developers to leverage the platform features in a controlled manner through predefined APIs. There are tools, both official and unofficial, which can also aid in the development of native apps. It is also common for developers to use frameworks in their native app to make development easier.

NATIVE APP ADVANTAGES

The native app comes with a number of benefits over the other types. The benefits revolve around being tightly integrated with the device platform.

- **Native APIs.** They can use the native APIs directly in the app, making the tightest connection to the platform.

- **Performance.** Native apps can experience highest levels of performance when they are designed well.
- **Same environment.** They use the same environment as the platform, which is helpful for developers familiar with the languages used.

NATIVE APP DISADVANTAGES

The disadvantages of native apps are generally the level of difficulty in developing and maintaining them.

- **Language requirements.** Requires proficiency in the platform language (for example Java) and knowledge how to use APIs.
- **Not cross platform.** Native apps can only be developed for one platform at a time.
- **High level of effort.** Typically native apps are more work and overhead to build, increasing costs.

Native apps may be best suited for developers who have a command of Java and Objective C, or for teams with extensive resources and a need for the benefits.

1.2.2 Mobile Websites (Web Apps)

Mobile websites are applications that work well on a mobile device, but are accessed through the mobile browser. Sometimes they are called Web Apps. Most simply, they are websites viewed on a mobile device in a mobile browser, with the exception of being designed to fit a mobile device screen size.

Some websites have a unique version of the normal website that have been developed specifically for use on a mobile device. Perhaps you've visited websites that redirect you on a mobile device to a limited feature application, often on a subdomain such as http://m.ebay.com. In other examples the design adjusts to the form factor and screen size in a technique called responsive design, such as http://www.bostonglobe.com. Depending on the site of the browser window, the website reflows the page to fit better on a smaller screen and perhaps even hides content.

MOBILE WEBSITES ADVANTAGES

Mobile websites enjoy a number of benefits, primarily in the level of effort and compatibility on devices.

- **Maintainability.** They are easy to update and maintain without the need to go through an approval process or updating installations on devices.
- **No installation.** Since it exists on the internet, it doesn't require installation on mobile devices.
- **Cross platform.** Any mobile device has a browser, allowing your application to be accessible from any device.

MOBILE WEBSITES DISADVANTAGES

Mobile websites run inside of a mobile browser, which is the major cause of limitations and disadvantages.

- **No native access.** Since it is run in the browser, it has no access to the native APIs or the platform, just the APIs provided by the browser.
- **Requires keyboard to load.** The user has to type address in a browser to find or use your mobile website, and mobile browsing is on the decline.
- **Limited user interface.** It is difficult to create touch friendly applications, especially if you have a responsive site that has to work well on desktops.

Mobile websites can be important even if you have a mobile app, depending on your product or service. Research shows users spend much more time using apps compared to the mobile browser, so mobile websites tend to have a lower engagement.

1.2.3 Hybrid Apps

A hybrid app is a mobile app that contains a web view (essentially an isolated browser instance) to run a web application inside of a native app, using a native app wrapper that can communicate with the native device platform and the web view. This means web applications can run on a mobile device and have access to the device, such as the camera or GPS features. Hybrid apps are possible because of tools that have been created that facilitate the communication between the web view and the native platform. These tools are not part of the official iOS or Android platforms, but are third party tools such as Apache Cordova, which is used in this book. When a hybrid app is built, it will be compiled, transforming your web application into a native app.

HYBRID APP ADVANTAGES

- **Cross platform.** You can build your app once, and deploy it to multiple platforms with minimal effort.
- **Same skills as web development.** It allows you to build mobile apps using the same skills already used to develop websites and web applications.
- **Access to device.** Since the web view is wrapped in a native app, your app has access to all of the device features available to a native app.
- **Ease of development.** They are easy and fast to develop, without the need to constantly rebuild to preview. You also have access to the same development tools used for building websites.

Hybrid apps provide a robust base for mobile app development while still being able to use the web platform. You can build the majority of your app as a website, but anytime you need access to a native API the hybrid app framework can provide a bridge to access that API by using JavaScript. You can detect swipes, pinches, and other gestures like you can detect clicks or keyboard events.

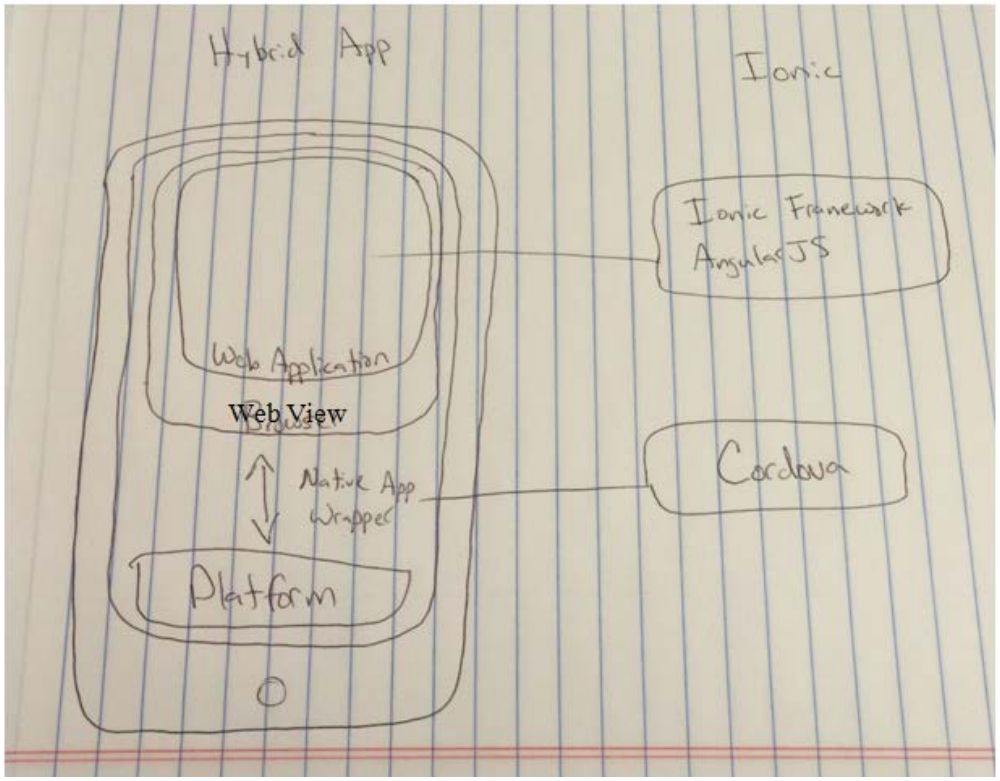
HYBRID APP DISADVANTAGES

- **Web view limitations.** The application can only run as well as the web view instance, which means performance is tied to the quality of the platform's browser.
- **Native via plugins.** Access to the native APIs you need may not be currently available, and may require additional development to make a plugin to support it.
- **No native user interface controls.** Without a tool like Ionic, developers would have to create all of the user interface elements. Luckily Ionic provides a rich set of controls that look and feel like the native controls.

With Ionic, we'll be building hybrid apps so we can leverage the knowledge and skills with which web developers are already familiar.

1.3 Understanding how the Ionic stack works

There are several technologies that can be used when building hybrid apps, but with Ionic there are three primary technologies: the Ionic Framework, AngularJS, and Cordova. Let's look at each one more closely.



How Ionic, Angular, and Cordova work together for a hybrid app

1.3.1 Ionic Framework

The Ionic Framework (often referred to as Ionic) is a number of things, but the primary feature is that it provides a set of user interface controls that are missing from HTML but are common on mobile apps. Imagine we are building a weather app that shows current conditions based on the user's location. Ionic provides a number of user interface components such as a slidebox that allows a user to swipe between several boxes of information like temperature, forecasts, and weather maps. These components are built with a combination of CSS, HTML, and JavaScript, and they behave like the native controls you are accustomed to using. Common examples include:

- Side menus that slide in from the side
- Toggle buttons
- Mobile tabs

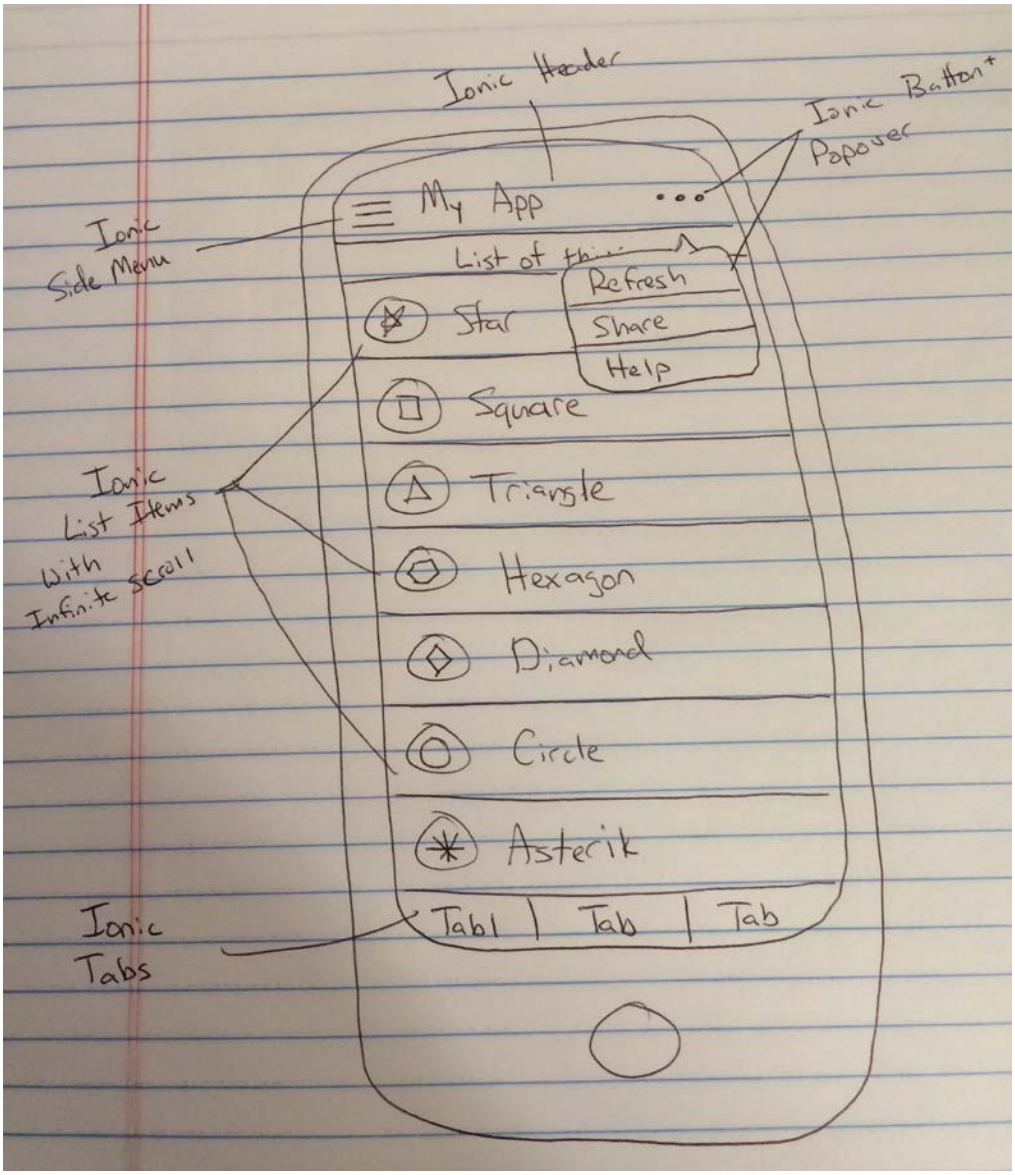


Diagram showing how parts of Ionic work together to create a useable interface

Ionic is an open source project that is primarily developed by Drifty. It was launched in November 2013 and has grown in popularity very quickly to become a primary choice when

building hybrid apps. Over 20,000 apps are launched with Ionic a month. Ionic is provided under the MIT license.

Ionic also has built a command line tool that provides some helpful developer tools. I'll refer to it as the Ionic CLI tool. This CLI tool can help generate starter projects, preview, build, and deploy your app. I'll demonstrate most of the features of the CLI tool as we go through our examples.

Ionic also has an included font icon library. This gives you access to a decent number of useful and common icons for your application. It is optional, but is provided by default and we will use it regularly in our examples.

Ionic also has a number of services in development that can aid in the development of apps such as a visual drag and drop app creator and deployment tooling.

These user interface controls are the primary feature of Ionic, but the Ionic team has worked hard to ensure its tools and processes work well with the following two projects, AngularJS and Cordova.

1.3.2 AngularJS:

AngularJS (often called Angular) is a Google open source project that has become quite popular with web application developers. It provides web developers the ability to write complete applications quickly and provides good structure for your applications. In our weather app example, Angular is used to help manage the user's data and loading information from our weather service.

Gone are the days where you have to use a server-based language (like PHP, Ruby, or Java) to build complex applications, today JavaScript web application frameworks like AngularJS allow you to build complex applications in the browser. This is an obvious advantage for hybrid app developers, since the browser is the platform we use to create our apps. If you are familiar with AngularJS (or other JavaScript application frameworks such as Ember) you will be able to apply your knowledge easily into mobile apps with Ionic.

Miško Hevery and Adam Abrons started AngularJS in 2009. Eventually Hevery joined Google and brought Angular with him. The project is immensely popular with developers today, and has been adopted by a number of large sites such as www.stackoverflow.com and www.nasa.gov. AngularJS is licensed under the MIT license.

In this book I will also use additional Angular modules that have been developed by third party developers. The primary example is a module called `ui.bootstrap`, which is an open source Angular module that provides better application routing and navigation compared to the default Angular routing module.

1.3.3 Cordova: Hybrid App Framework

In this book, we will be using Apache Cordova as our hybrid app framework. This is the layer that takes care of managing the communication between the browser window and the native APIs. Our weather app example needs access to the device's GPS information to know what

location to load data for, and Cordova is able to bridge the gap between Angular and the device to retrieve that information.

The core of Cordova provides a lot features, and it also provides a plugin system for developers to create new features such as native API integrations with the phone camera. It is actively maintained and regularly releases new version with improvements and new features. You can find out more about Cordova at http://cordova.apache.org. Later in chapter 11 I'll cover more detail about Cordova and plugins.

You may also have heard of Phonegap. Adobe contributed Phonegap to the Apache Software Foundation under the name Cordova. Today, Phonegap is a distribution of Cordova, or in other words Phonegap is essentially Cordova with support for a few additional commercial features from Adobe. For the purposes of this book we will use Cordova, but you could use Phonegap and its commercial features if you desire.

Cordova is an open source Apache project that has a large community around it. Adobe continues to be a major developer of the framework. Cordova is licensed under Apache 2.0 license.

1.4 Why Ionic?

Hybrid app frameworks are not new, there are a number of options available, but Ionic brings a new and important set of improvements. Until recently, mobile devices were still relatively sluggish and only a native app could deliver the performance and experience many developers wanted or needed. Mobile platform makers had not made browsers as fast as the native platforms. All of that has changed as devices have become more powerful, platforms have improved, and new tools like Ionic make it possible to build amazing hybrid apps.

1.4.1 Why Ionic is good for developers

Ionic is able to provide an experience built into the hybrid app that looks, feels, and performs like a native app. The long-standing argument that native apps are the only way to get fast and richly featured apps has been proven wrong. People expect their mobile apps to be fast, smooth, and intuitive, which Ionic apps can provide.

- **Build apps with the web platform.** Using HTML, CSS, and JavaScript, you can make hybrid apps that behave like native like mobile apps.
- **Built with AngularJS.** For people familiar with AngularJS (or even another JavaScript framework like Ember), Ionic is a perfect choice. Ionic is built alongside Angular, which allows you access to all of Angular's features as well as any of the many thousands of Angular modules for additional features. Ionic also uses the popular `ui.router` module for navigation, because Angular's default routing module lacks some features.
- **Uses modern techniques.** Ionic was designed to work with modern CSS3 features, like animations. Mobile browsers generally have better support for the latest web platform specifications, which allows you to use those features as well.
- **Powerful CLI tools.** With the Ionic command line tool, you can quickly manage development tasks such as previewing the app in a browser, emulating the app, or

deploying an app to a connected device. It helps with setting up and starting a project as well.

- **Ionic ecosystem.** Ionic also provides a rich ecosystem of features that make development much easier. The Ionic Creator service allows you to use a drag and drop interface to design and export an app. An upcoming service for remotely building and deploying apps is also in development. In short, Ionic is all about creating not just the basic tools for making hybrid apps but also the development tools that will help you create them efficiently.

You could even consider just using the parts of Ionic that you want, such as the UI components for a mobile app but use a different hybrid app build tool. While Ionic has a lot of features that work well together, you can select the parts you want to use and apply them to an existing project or workflow.

1.4.2 Why Ionic is good for managers/owners

When I began to use Ionic I had to make the argument about why it was the right choice over other options. In a short presentation I made the following case for Ionic.

- **Ionic covers all the bases.** Ionic was able to provide a complete set of features needed to build a hybrid mobile app, with the current developer team skills available.
- **Ionic is open source.** Many developers are very fond of open source, but managers should also be interested in it. Ionic has permissive open source licensing that doesn't get in the way of our own copyrights.
- **Ionic has a dedicated team.** Open source projects can be difficult to select, because you can't be sure if it will be properly developed or supported. Ionic has a dedicated team that has a vested interest in keeping the platform on the leading edge.
- **Ionic development is fast.** Within minutes you can generate a base app and very shortly build a solid prototype. This is important to time management, costs, and maintainability.
- **Ionic is fun.** Making your developers happy is pretty important to their productivity. This is a bit more subjective statement, but the developers I know who have used Ionic enjoy it.

Selecting Ionic is a good move for managers who want to ensure they have a solid platform that is well supported and their developers can quickly leverage to build mobile apps.

1.4.3 Why Ionic is good for your customers

Ionic provides developers the foundation and tools needed to build up great apps. Building on the benefits above, it allows hybrid apps to offer:

- **Native experience.** With Ionic, you can create a look and feel that is like the native apps, making it easier for your customers to use the app.
- **Performance.** The performance with Ionic is comparable to a native app, and the better

the app performs the happier customers will be.

- **Beautiful design.** Customers are accustomed to having apps that look great, and with Ionic as the base you can create an amazing experience.

With Ionic, you can design amazing apps that your customers will be delighted to use and take you far less time and effort to create.

1.5 Prerequisites for building apps with Ionic

In order to build hybrid apps, there are a few skills that you should already have which are not covered in this book. You don't need to be an expert in any of the following areas, but you should be prepared to use them all together.

1.5.1 Experience with the Web Platform

If you've built a website, you've used the Web Platform. The browser is like the operating system that we will be using to develop our mobile app. HTML, CSS, and JavaScript are the key languages the browser understands. HTML gives structure to the content, while CSS provides the design. JavaScript then provides for the interaction and logic necessary for our web application.

jQuery is often used in web development, but in this book we will be using raw JavaScript. You will need to be familiar with JavaScript syntax and concepts such as asynchronous calls and events.

1.5.2 Experience with web applications and AngularJS

You should have a fundamental understanding of web applications, since we will be building them inside of our mobile apps. There are a number of technologies and libraries that developers use to build web applications, and familiarity with the concepts will help you greatly.

In this book, our web applications will be written in JavaScript using the popular AngularJS framework. Ionic is built specifically to work with Angular, and developers who have experience building applications with Angular will be able to apply their experience easily. You might have experience with another framework, such as Ember or Backbone, which can provide a foundation as you learn the Angular specific approach.

I will cover a bit about Angular as we go since it is used heavily, but this is not a book about Angular. You'll want to refer to Angular In Action to learn everything you want about Angular beyond the scope of this book.

1.5.3 Access to a Mobile Device

Having a mobile device is extremely important when building a mobile app. I recommend that you have at least one device for every platform in order to test on an actual device. There are emulators that let you see what your apps should look like on a mobile device, but they are not full substitutes for the real thing.

You will have to register these devices with your developer accounts as well, so it's not practical to borrow. If you need a device, you can check for refurbished or used items online and use it just for development testing. The more types of devices you can test your app with, the better.

These three prerequisites will help you be more successful at designing, testing and building mobile apps across multiple platforms. Let's take a look at the mobile platforms Ionic supports.

1.6 Supported Mobile Devices and Platforms

There are a number of mobile platforms: iOS, Android, Windows 8, Firefox OS, Tizen, Blackberry, and more. With Ionic, we can build for both iOS and Android. Support for Windows 8 and Firefox OS are planned for the future, but not currently available.

While it may be possible to develop an app by previewing only on a simulator, devices can act differently in the real world. When possible, it is recommended to have an actual device available for the platform to which you are building. It also helps to test apps on a device because you are able to experience how the app responds to touch gestures that are difficult to simulate. Let's take a closer look at these two primary platforms and requirements.

1.6.1 Apple iOS

Apple makes the popular iPhone and iPad devices and they share a common platform called iOS. Apple has strong control over the entire experience from the devices to software to the apps, essentially making it a closed system. This has made iOS a strong platform from the perspective of users and developers.

Apple provides Xcode as the primary development program for iOS and Mac development. Xcode is free and available in the AppStore if you do not already have it downloaded. I will cover setting up for iOS development in the next chapter.

Xcode comes with a set of simulators, which allow you to simulate different versions of their iPhones and iPads. The simulators are fairly good at giving a realistic experience, which is helpful when targeting multiple versions of iOS with the same app.

Apple has one major requirement for building iOS mobile apps: you will need a Mac computer. Apple has only designed its development tools to work on Apple's operating system OSX, and it is also recommended to be running the latest version. Xcode is the program used to manage mobile app development, which is provided for free by Apple, but only runs on OSX.

For those of you who aren't using a Mac, it is worth considering purchasing one if you plan to do iOS development. If you just need to build mobile apps, you'll be able to take advantage of any of the Mac computers. Any new Mac will have enough processing power to manage the simulation and build process. If you consider purchasing a used machine, you should verify that it is able run the latest version of OSX.

If you don't have a Mac, there are some options available that can help build your apps. (Possible Ionic tool here, need to do more research into options. Most are only able to provide a build, not simulators.)

The Apple Developer Program has two types of membership, iOS and Mac development. You will need to sign up at http://developer.apple.com and join the iOS program. It costs $99 USD per year, but you only need to sign up when you are ready to sign and deploy your app to the AppStore. You are able to work through this entire book without this account until the point I show you how to deploy an app into the AppStore.

1.6.2 Google Android

Google created Android as an open source mobile platform, and has allowed mobile device makers to take Android and integrate it into their devices. Compared to Apple's approach, Android has a very diverse set of devices. Older devices may also have Android forks specifically designed for a mobile carrier. This open system has encouraged adoption and also has been the leading platform in emerging markets, by allowing lower cost devices due to the absence of licensing fees for the operating system.

Android provides a number of tools for developing, which are freely available for download from Android's site. Google has also been working on additional tools that are being built into Chrome, Google's browser, providing useful development support for hybrid app developers. I'll cover how to setup your computer for Android development in the next chapter. Android has a simulator that can emulate the screen size and resolution of most Android devices.

Android development is supported on Mac, Linux, and Windows computers. You can review the exact requirements for Android developer tools https://developer.android.com/sdk/index.html.

Google also has a Developer Program, which is $25 USD per year. Just like with iOS, you don't have to sign up until you are ready to publish your app into the Play Store. You can register at https://play.google.com/apps/publish/signup/.

There are a few other Android app stores, notably the Amazon Web Store, which may also charge for a developer program. These are not covered in this book. However you will be able to build and deploy apps for any Android based device, even if the app is distributed through a different store.

1.7 Summary

Through this chapter we've looked at details about how Ionic provides a powerful set of tools for building hybrid apps. Let's review the major topics covered in this chapter.

- Ionic is a solid choice that benefits developers, managers, and users.
- Hybrid apps are an advantage for developers who already are familiar with the web platform, and do not require learning additional programming languages.
- Hybrid apps use a web view inside of a native app to run web applications that have access to the native APIs.
- Ionic is designed to work with AngularJS for web application development and Cordova for integration with the device platform.
- Android and iOS are supported and both require developer subscriptions. iOS can only

be developed from a Mac.

2

Build Your First App

In this chapter you will learn how to:

- Setup your computer for Ionic and dependencies
- Create a working sample project
- Preview an app in a simulator
- Build an app to a connected device

You are probably ready to get started with some code and actually building a mobile app. We're going to walk through the steps to get a new project up and running. By the end of the chapter you will have a sample app running on your computer and when you are ready you can setup and preview on a connected device. The steps in this chapter will be applied to future chapters, so you may find it a useful reference.

This chapter has two parts for setting up your development environment. The first part is a quickstart guide to getting the basics installed and running with a sample app and previewing it in a browser. This is great for getting up and running and being able to develop quickly. Think of this like setting up the development environment. The second part is a guide to setting up ways to view your app on an emulator or on a connected device. This is like setting up the previewing environment. If you are rearing to go, feel free to skip the second part of the chapter and come back later. The emulator and connected devices are not necessary until you are ready to test your app in a mobile environment.

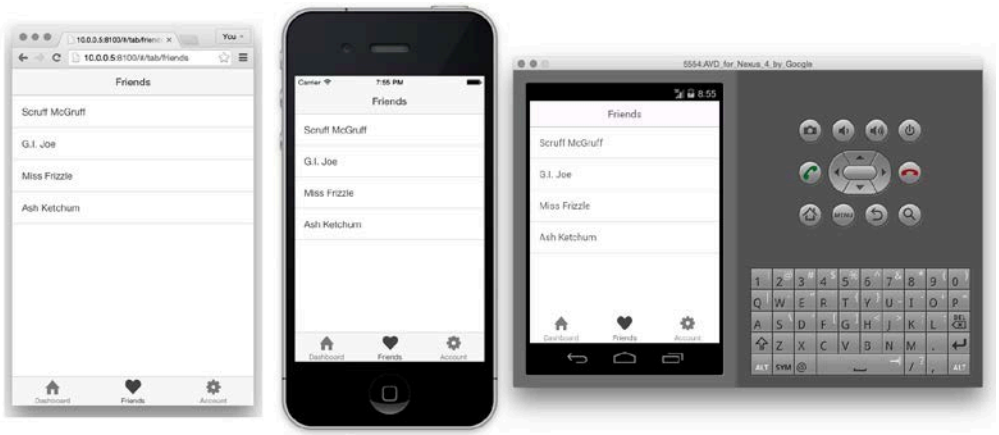


Figure 2.1 You will be able to preview a sample app in a browser, an emulator, and with a connected device.

In this book the command line will be your friend. On Windows you'll use the Command Prompt which can be found in the program list or opened by running `cmd.exe` from the Run bar. On Mac you'll use the terminal, which can be found in the Launchpad or by typing terminal in Spotlight. I recommend adding a shortcut on your desktop for Windows or adding it to the dock in Mac since it will get a lot of use.

2.1 Quickstart Guide

In this section we will get the essential development environment setup, setup our first app, and preview it in a browser. Since we are building a hybrid app, the browser is the easiest way to preview our app.

The majority of your development time will be using the browser for previewing and developing your app. As your app matures, you will likely begin to use an emulator to simulate a real mobile device. Finally as the app is nearly complete, you will want to deploy it to a mobile device. Figure 2.2 shows a typical workflow during development, and this section covers the browser preview while the next section covers the other two options.

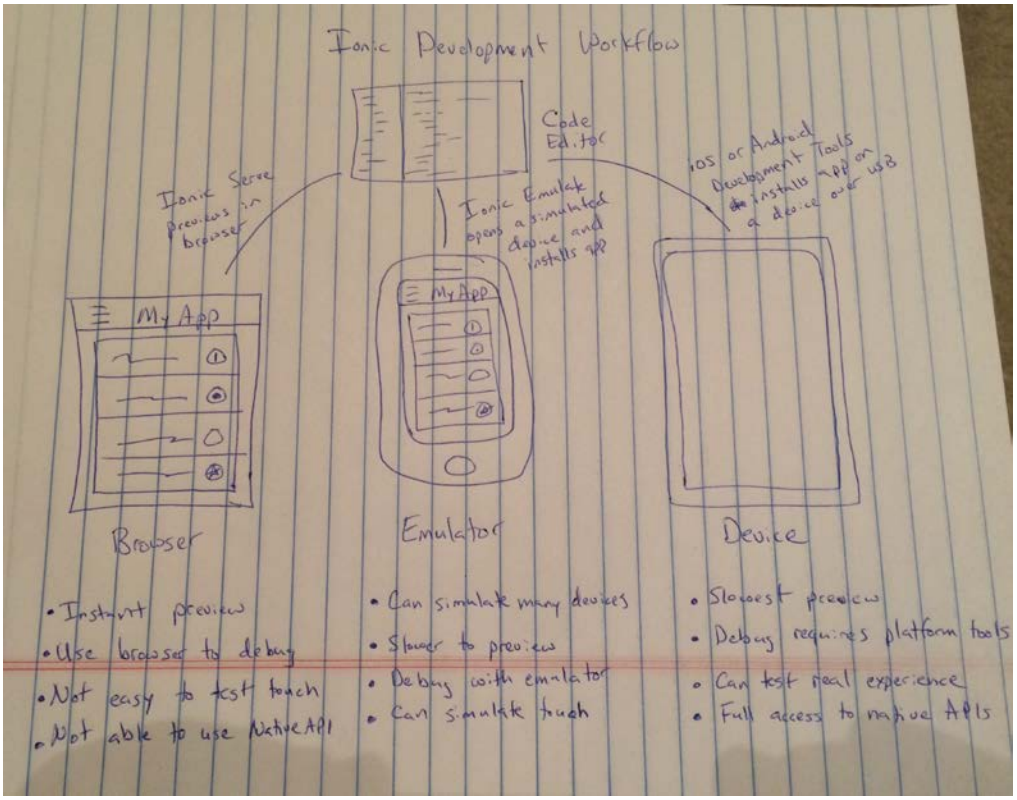


Figure 2.2 Typical workflow, and reasons why you would preview in a browser, emulator or device.

2.1.1 Setting up your development environment

To begin building mobile apps with Ionic, there are some dependencies required. I'll walk through how to install and setup those dependencies on your computer.

In the following table 2.1, you can see the list of software you need to have installed on your machine to get started.

Table 2.1: Software required for your development environment

Software	Homepage
Git	http://git-scm.org
NodeJS	http://nodejs.org
Ionic	http://ionicframework.com

Cordova http://cordova.apache.org

If you already have these installed, you can jump to the next section. Otherwise, let's get each of these installed.

INSTALL GIT

Git is a popular and powerful source code management system. If you aren't familiar with git, you can review some of the very helpful documentation from GitHub (the most popular git repository service) at https://training.github.com. Git is required to be on your computer for some of the tools to work, so we need to make sure it is installed and available through the command line.

First we can check if git is available by running the following command.

```
$ git -v
v2.0.4
```

If you don't get a version number, you will need to install git. Visit http://git-scm.org to find the appropriate download and latest version. You may also want a git client for a nicer interface. There are a number of options out there, both free or paid, but in this book I'll always use the command line.

INSTALL NODEJS

NodeJS (often referred to as Node) is a platform that runs JavaScript outside of the browser. It allows developers to create applications written in JavaScript that can then execute anywhere. Ionic and Cordova are both written on top of NodeJS, so it is the first requirement to have installed.

Node can be installed on your machine by going to http://nodejs.org and downloading the package for your platform to install. If you already have Node installed, you should go ahead and install the latest stable version.

You can validate that Node installed correctly by opening a terminal in OSX or the command prompt on Windows and executing the following to check the version of Node.

```
$ node -v
v0.10.30
```

If you have any issues with installing Node, you can review the documentation on the Node website. Now we'll use Node's package manager to install Ionic and Cordova for us.

INSTALL IONIC CLI AND CORDOVA

We can easily install Ionic and Cordova in a single command. This uses the Node package manager (npm) to install and setup our command line tools. Make sure git is already installed first!

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

This may take a few minutes, depending on the speed of your connection. Ionic and Cordova will be installed in such a way that they are available from the command line. Both of these tools execute using Node, but are aliased so you can run them with just the `cordova` or `ionic` commands. Test that they are correctly installed by running the following commands and ensure they execute without errors.

```
cordova –v
ionic
```

Setting up your development environment is important, so make sure each of these are installed and up to date. You should keep Ionic updated, and it will alert you when updates are available. You should update Cordova when there are new features you need or bugfixes. Sometimes updating Cordova may require updates to your project, so it should be done only when necessary and always review the Cordova documentation about possible required changes. To update Ionic or Cordova, you can run the following commands.

```
$ npm update –g ionic
$ npm update –g cordova
```

At this point we have everything we need, so let's start setting up our sample app.

2.1.2 Starting a new project

Ionic provides a simple command that allows you to setup a new project in seconds. Ionic provides a set of starter templates that you can use to begin. We will use the sidemenu template. Run the following command to create a new project

```
$ ionic start chapter2
```

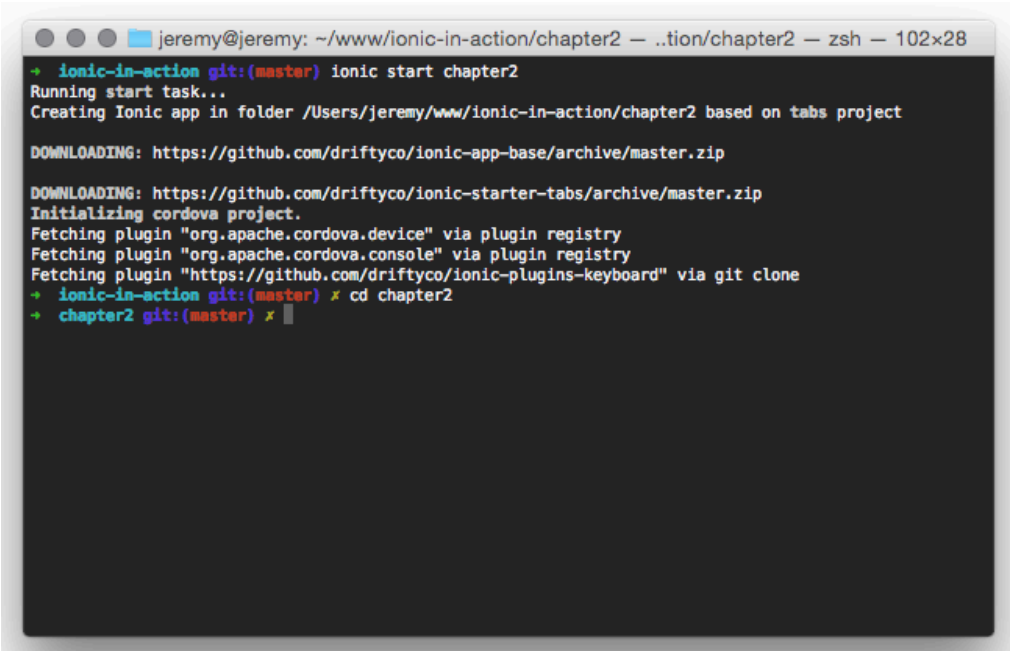


Figure 2.3 Using the ionic start command will generate a simple project scaffolding

Ionic will create a new folder called `chapter2` which will be used to setup the new project using the `tabs` template. Let's take a moment to understand what each folder is for.

Ionic Command Line Utility

There are a number of commands available with the ionic utility. To see the available commands you can look at the help details by running `ionic --help`.

To see more details about what the utility can do and more documentation, you can view the source code on GitHub at https://github.com/driftyco/ionic-cli.

2.1.3 Project folder structure

The project folder contains a number of files and directories, which all have a unique purpose. Here are the files and directories you should see in a new project.

```
.bowerrc
.gitignore
README.md
bower.json
config.xml
gulpfile.js
```

```
hooks
ionic.project
package.json
platforms
plugins
scss
www
```

This is the generic structure of any Ionic app. The files and directories that are setup and required by Cordova are `config.xml`, `hooks`, `platforms`, `plugins`, and `www`. The rest are created by Ionic. Ionic uses both Bower and NPM to load some dependencies for the project. If you aren't familiar with them, you can visit information about Bower at http://bower.io and NPM at https://npmjs.org.

The config.xml file is used by Cordova when generating platform files. It contains the information about the author, global preferences, platform specific preferences, plugins enabled, and more. The default config.xml file generated will use Ionic as the author and HelloWorld as the app name. You can read about all of the options here https://cordova.apache.org/docs/en/edge/config_ref_index.md.html.

The www directory contains all of our web application files that will be run inside of the web view. It is assumed that there will be an index.html inside, otherwise you could structure your files however you like. By default, Ionic starts you with a common scaffolding for AngularJS type web applications.

We'll cover these files and directories in more detail as they are used. Now that we've got our files generated, let's preview our sample app.

2.1.4 Previewing in a browser

We can preview our app in the browser, which makes it very easy to debug and develop without having to constantly build the project onto a device or emulator. Typically you will develop your app using this technique, and then test in the emulator and on a device when the app is more complete. The following command will start a simple server, open the browser, and even auto-refresh the browser when you save a file change.

```
$ ionic serve
Running serve task...
Running dev server at http://0.0.0.0:8100
Running live reload server at http://0.0.0.0:35729
```

It will open the default browser on your computer on port 8100. You can visit http://localhost:8100 in any browser, but it is best to preview using a browser used by the platform you are targeting since that is the browser the web view uses.

Since you are viewing the app in a browser, you have access to the developer tools you would use for building websites. As you develop, you will want to have the developer tools open to aid in development and debugging as you see in figure 2.3.

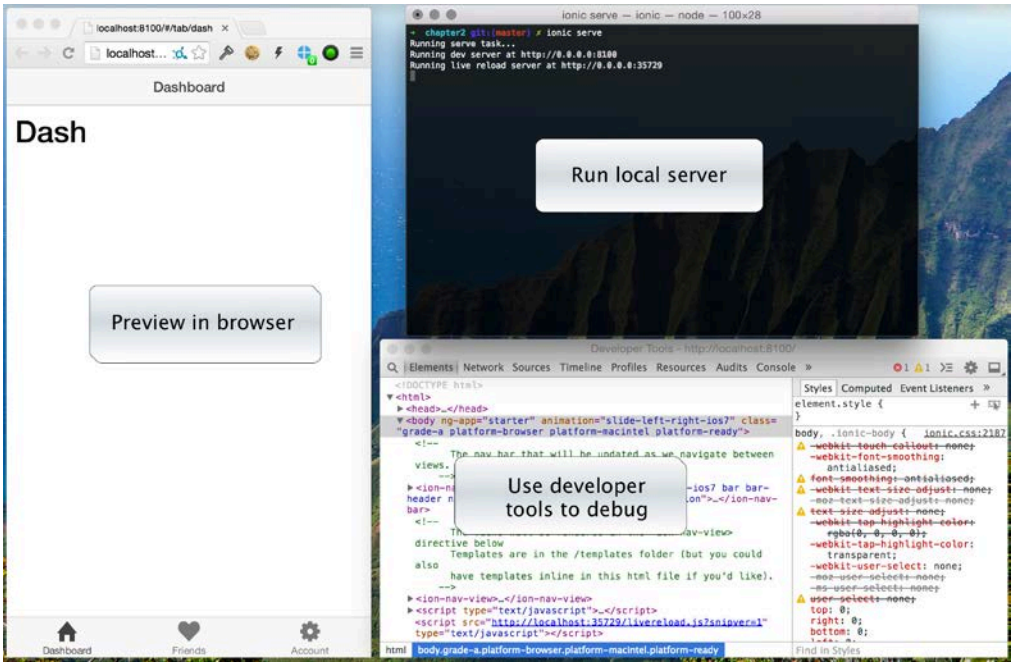


Figure 2.4 Previewing an app in the browser gives you access to the browser's developer tools.

Does it matter what browser I use to preview?

While you are free to use any browser for previewing your app, you should really consider using Chrome or Safari. The main reason is that since iOS uses Safari and Android uses the Android Browser (though it may change to Chrome), using the same browser on your computer will simulate the mobile environment most effectively.

The browser on your mobile device, and used in the web view, is not identical to the browser on your computer. However, they are certainly related and tend to support the same general features.

Safari for Windows should not be used for previewing, as it is no longer supported.

2.2 Setting up previewing environments

This section will guide you to setup both emulators and connected devices for previewing your mobile app. Both allow you to preview the app like it is intended to be used, not just in a browser but inside of a mobile device. An emulator is a virtual device, which actually runs the mobile platform (Android for example) in a container and can execute your app like a real physical device. A connected device is any physical device that you connect directly to your computer with a USB cable, and you are able to install your app directly onto it.

To get everything setup, we need to do the following.

- Install platform tools needed for building apps
- Download and setup emulators for previewing
- Setup a connected device for previewing
- Setup the project for each supported platform and preview

2.2.1 Installing platform tools

We need to install additional software in order to emulate and deploy to connected devices. You only need to setup the software for the platforms you wish to support. The following table 2.2 has the required software for Android and iOS development

Table 2.2 Android and iOS software for emulating and deploying to devices

Platform	Software	Where to install
iOS	Xcode	Search for 'Xcode' in AppStore
Android	Android SDK	https://developer.android.com/sdk/

OSX ONLY – INSTALL XCODE FOR IOS

Apple requires Xcode for the emulation and distribution of iOS apps. It is only available for Macs, so if you plan to support iOS then you'll need to have a Mac.

You can download Xcode by opening the AppStore and searching for Xcode. It is an official Apple app, and it is quite large so be sure to have enough free space.

Figure 2.5 Xcode is free and available for download through the AppStore on your Mac computer.

INSTALL ANDROID DEVELOPER TOOLS

Android development can be done on any Windows, Mac, or Linux. Android runs Java, which is cross platform as well. Android provides three options to choose from. You can download the tools from http://developer.android.com/sdk. I will use the Standalone SDK in this book, because it uses the command line and is more concise. You can choose the IDE options, both also include the SDK, but their use is not covered in this book.

- **Standalone SDK**: This is just the base tools required. They are bundled into the other two options, but lack any integrated environment. If you already have a preferred development environment or don't wish to install the packages below, you must install this.
- **Eclipse with ADT**: As the first Android developer environment, it is the current toolkit for Android developers. It has some age to it, but is a functional way to develop for Android. It will be retired for the next option in the future.
- **Android Studio**: The evolution of the Android developer environment, based on IntelliJ, has a more advanced set of features for improved development workflows. It is in beta, meaning eventually it will replace the Eclipse with ADT environment as the default and preferred Android developer toolkit.

When you install the Standalone SDK, make sure you add the directory to your path so you can easily execute Android commands. To verify the installation was successful, you can run the following command to see the android help.

```
android –help
```

If the command fails, then you need to make sure the SDK is available in your system path.

2.2.2 Setting up emulators

Emulators allow you to run a virtual device on your computer that simulates the actual environment of a mobile device. It will run the platform inside of the emulator, so for example inside of an Android emulator you can run the actual Android operating system and install your app for development.

You will want to use an emulator when you are ready to test various types of devices quickly or you need to test your app on a device you do not have access to. It is slower to preview in an emulator compared with the browser, so you will likely emulate when your app is already functional in the browser.

The emulators are not automatically setup, and can require some time to download and setup. Let's learn how to setup both Android and iOS emulators.

SETTING UP AN IOS EMULATOR

Emulators are referred to as simulators in Xcode. To begin setting up your iOS simulator, launch Xcode and open the Preferences. In the Downloads tab, you will see a list of availabile optional packages, which include documentation and iOS simulators.

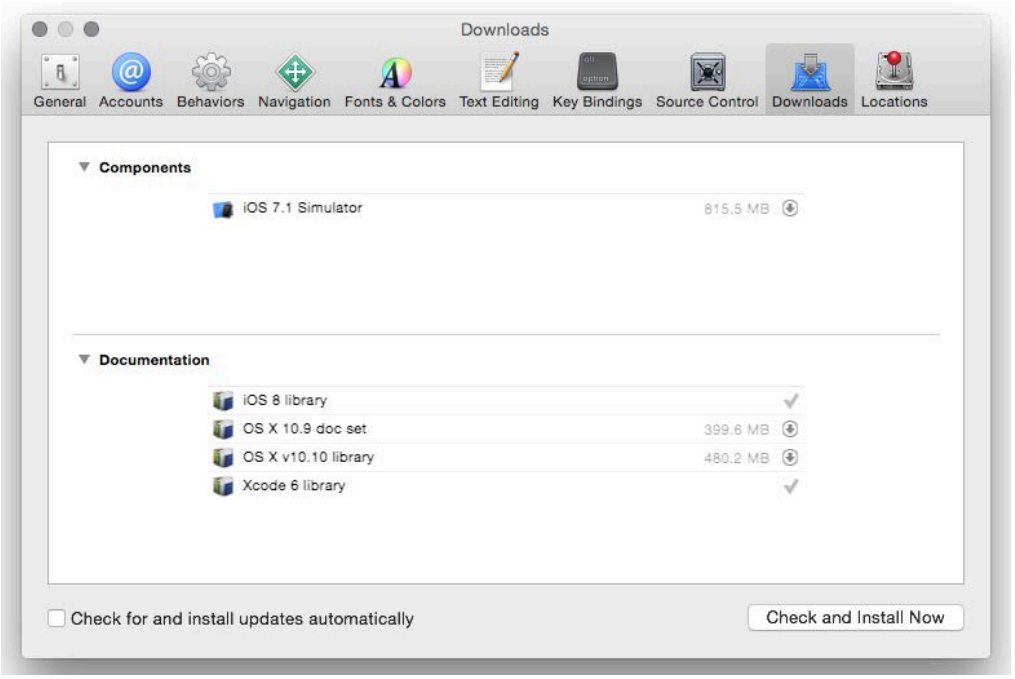


Figure 2.6 In Xcode Preferences, use the Downloads tab to download and install iOS Simulators.

I would suggest downloading only the most recent simulator at this point. Later you can install the emulators for all versions of iOS that you plan to target for testing. The documentation is also not necessary, as you can find it all online should you need it. Since these simulators and documentation are very large, save yourself the time and disk space and download only what you need.

Once the download is completed, your iOS simulator will be setup and ready to use. You can reset the emulator if you ever need to by having the simulator open and going to the iOS Simulator in the top menu, and selecting Reset Content and Settings.

What versions of Android or iOS should I use?

Ionic provides support for iOS6+ and Android 4+ (with limited support for Android 2.3). Generally it is a good idea to target the lowest version possible to increase the potential use base. Setting a minimum version number will prevent devices running older versions from being able to install your app. However, there may be reasons to limit support to newer versions if your app uses additional plugins or features that aren't available on older versions.

2.2.3 Setting up an Android emulator

Android emulators are much more free form compared to iOS emulators because they allow you to build your own device by declaring the device specifications. Luckily there are some presets that help guide us through this process, but due to the wide variety of Android devices available it is a bit more complex than for iOS.

We need to setup the SDK packages, so run `android sdk` in the command line. The SDK Manager will appear. It allows you to download the platform files for any version of Android, which is a bit more than we need. For the time being, it is recommended to download just the most recent release packages and core tools. You need to select the following items.

- Tools
    - Android SDK Tools
    - Android SDK Platform-tools
    - Android SDK Build-tools (version should match API used below)
- Android 4.4.2 (API 19)
    - SDK Platform
    - ARM EABI v7a System Image

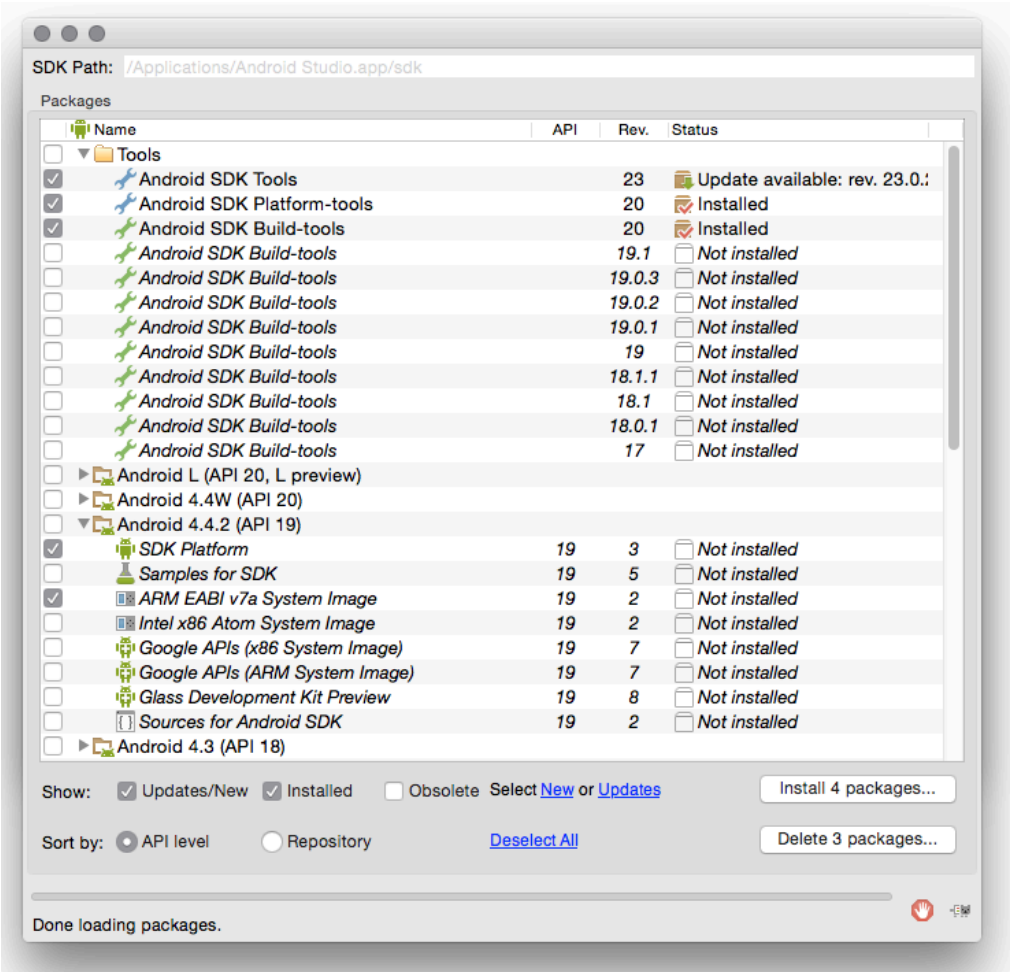


Figure 2.7 Select the SDK Tools, Platform-tools, and most recent Build-tools packages as well as the most recent stable release of Android SDK Platform and the ARM System Image.

Now we have to define an emulator device specifications. This gives us control over the exact device features such as RAM, screen size, and so on. Open the AVD (Android Virtual Device) Manager by executing the following command.

```
android avd
```

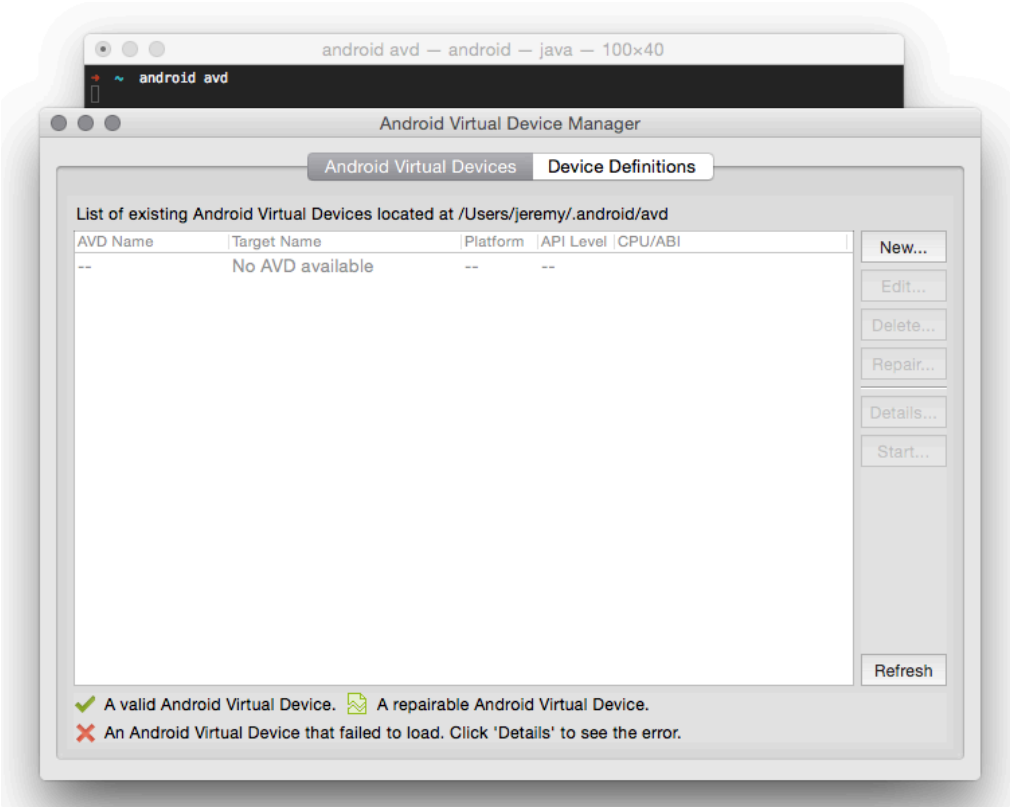


Figure 2.8 Open the Android Virtual Device Manager with the command `android avd`.

We have no devices setup, so select the Device Definitions tab so we can setup a device based on a known device configuration. I recommend using a Nexus 4 or Nexus 5 device, since they are developed by Google.

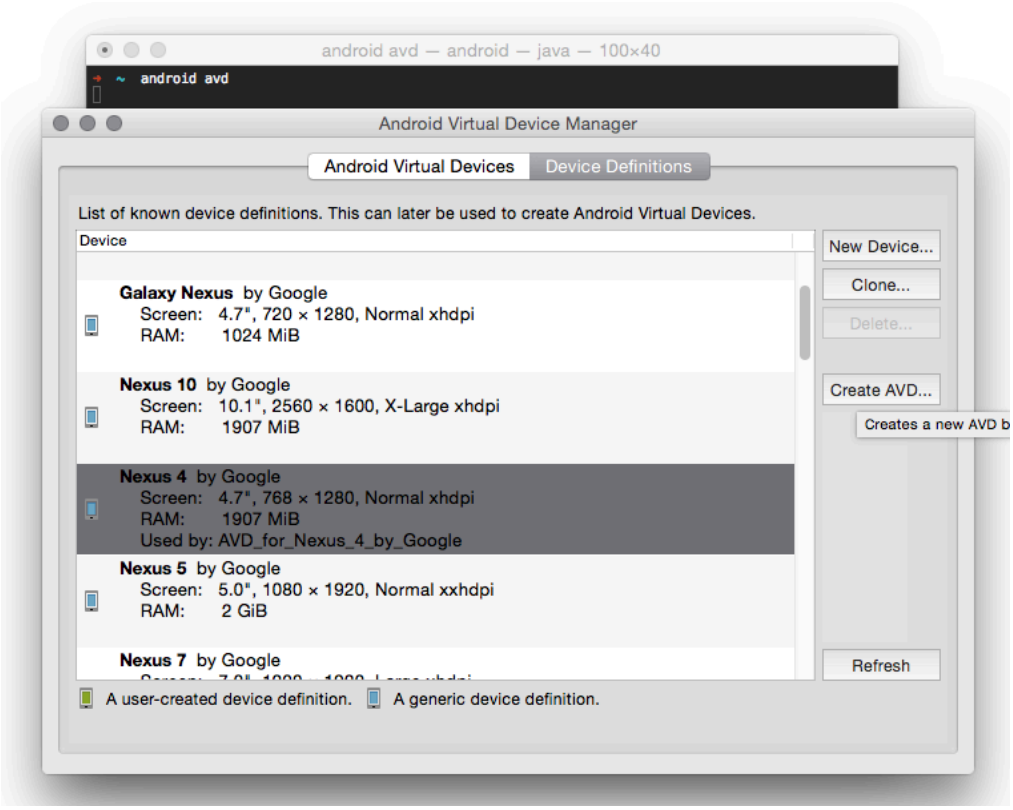


Figure 2.9 Find and select a device definition that you want to base your configuration from, and select Create AVD.

Once you've selected the device from the list, select Create AVD and it will open a form with additional details that you can specify about the device. Here you can decide what version of the Android platform is run, screen size and resolution, and more. Select the presets like in the following figure.

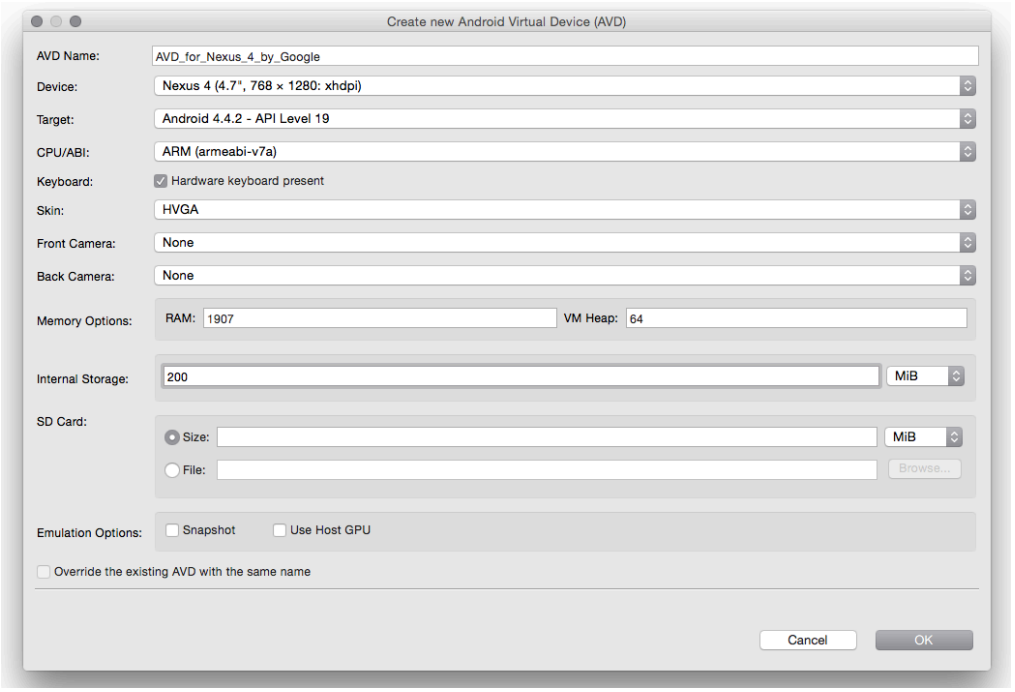


Figure 2.10 Create a new device based on the Nexus 4 by Google. This device has the same basic features as the actual phone, though we haven't enabled cameras.

Once you've finished, click OK and it will save your device. You can create or delete devices as needed, just make sure to always have one setup for emulating. The first time it runs it may be a bit slow since it will have to do some extra things to setup and boot.

Now you've got an Android device setup, we can reuse it in our projects

2.2.4 Setting up a connected device

If you have a device, you will want to be able to connect and deploy your apps to it. You can setup any number of connected devices that you have access to, in case you have both new and older devices that you want to be able to test. You'll want to test on devices when possible before you attempt to deploy to a store, and whenever you need to verify the functionality behaves as you expect with a touch environment.

SETTING UP AN iOS DEVICE

In order to connect your iOS device and deploy your app, you have to have an Apple Developer account with iOS. You will connect your iOS device to your Mac and open Xcode. Select Window and then Organizer from the top menu to open the Devices manager.

Apple requires security profiles to be setup so that your phone is verified to be connected for deploying your apps. You must connect your account in the Accounts tab of Preferences,

and it will help you setup certificates and provision profiles. Xcode should guide you through the steps for your device, as they may vary. For additional assistance, refer to Apple's documentation here https://developer.apple.com/Library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/RunYourApp/RunYourApp.html - //apple_ref/doc/uid/TP40010215-CH17-SW6. Once the profiles are setup, your device should be available to deploy.

SETTING UP AN ANDROID DEVICE

The first step is to enable developer settings on your Android device. In the Settings, you will want to access the Developer options. On Android 4.2 or greater, open the Settings > About phone (or tablet) view and tap on the build number 7 times to activate the Developer options panel, otherwise it is hidden from you. Enable USB Debugging from the Settings panel and connect your Android device to the computer via USB.

2.2.5 Adding a platform to the project

Before we can preview our app in an emulator or on a device, we need to setup the project to support the platform(s) of choice. Again we open the command line to use the Ionic tool.

```
$ ionic platform add ios
```

You can add only one platform per command, so if you plan to support multiple platforms you will have to add them individually. You can see how each platform triggers a different set of tasks that are required to setup the project for that platform.

Inside of the platforms directory, there will now be a new folder for each platform added with platform specific files inside. Currently they are just scaffolding files, but eventually they will be used to generate the actual app files.

2.2.6 Previewing in an emulator

Now that at least one platform has been added to your app project, we can use the one of the platform's emulators to preview our app. If you haven't already setup an emulator, you will need to do that first. Emulators are great for testing in a more real world environment, but are slower to use during development. Launching and previewing in an emulator takes some processing time to setup and begin, especially the first time.

```
$ ionic emulate ios
```

The emulator should open after running a number of tasks. You'll see a lot of output in the command line as it builds and generates the necessary files, but as long as it ends with a success message the emulator will launch and load your app.

When emulating Android, you can use `--target=NAME` in order run the app in a specific device you created, otherwise the default emulator is used. iOS lets you change the hardware once the emulator has opened from the Hardware top menu.

If 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 relanching it every time you change files.

2.2.7 Previewing on a mobile device

Nothing beats the real thing. If you have an Android or iOS device, you will most likely want to deploy your app on it at some point. While it is very useful, it is also slow and more difficult to debug. If you haven't

PREVIEW ON AN IOS DEVICE

In your project, make sure you've added iOS as a platform and navigate to the `platforms/ios` directory and open the file with the extension `.xcodeproj`. This opens the Xcode project for your app, and you can then select the device as an option to deploy.

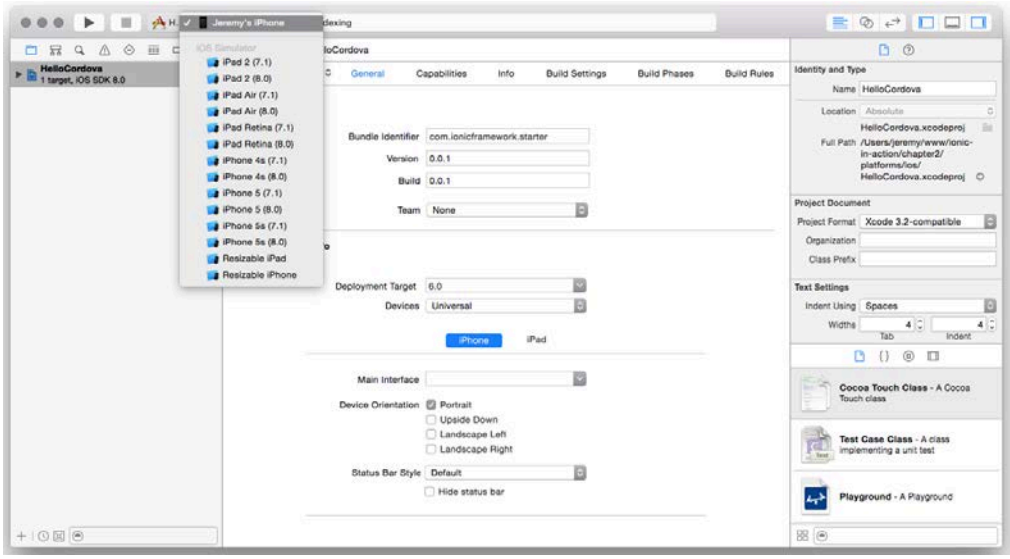


Figure 2.11 You can select the device or emulator that you wish to deploy from Xcode by opening the project.

You can deploy that app to your phone as many times as you want, as long as it has the same identifier it will override the existing version. You can also uninstall the app just like any other app by holding on it until the shakes, and tapping on the X in the corner.

DEPLOY TO AN ANDROID DEVICE

As long as you've already added Android to your project, you can deploy to a connected Android device with a few steps.

First, you need to manually add an attribute to the `platforms/android/AndroidManifest.xml` file. Android requires you to declare your app as debuggable. On the `<application>` node, add the following attribute.

```
android:debuggable="true"
```

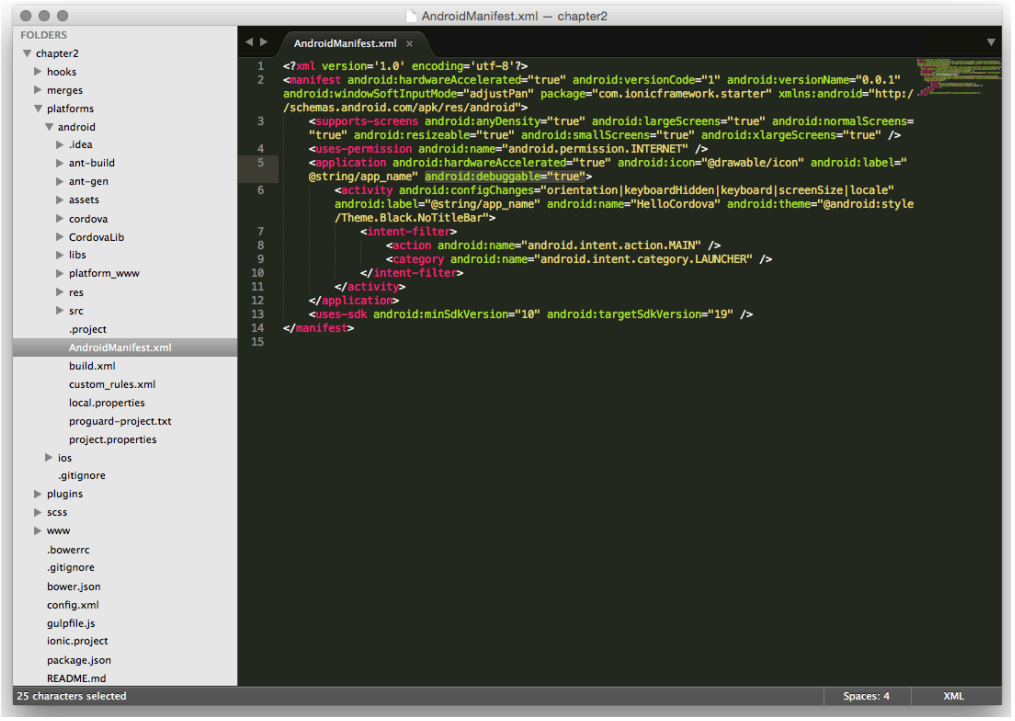


Figure 2.12 Adding android:debuggable="true" to the AndroidManifest.xml file enables the app for deployment to a connected device.

Ensure USB Debugging is enabled, and if it isn't you can refer back to 2.3.2 Setting up an Android device.

If you are on Windows, you will need to download the appropriate USB driver for your device from https://developer.android.com/tools/extras/oem-usb.html. If you are on a Mac, you don't need to do anything. If you are on Linux, you should consult the steps on https://developer.android.com/tools/device.html.

To confirm the device is connected, run adb devices from the command line. You should see a list of devices, and if you've setup any emulators they should appear as well.

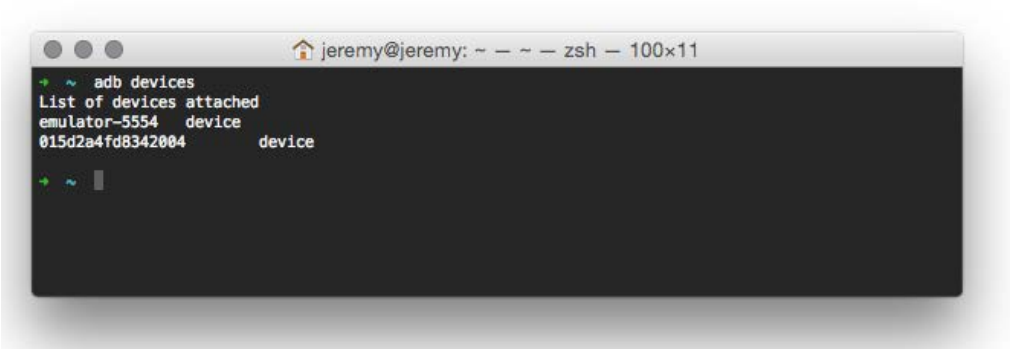


Figure 2.13 Android devices listed by running 'adb devices' from the command line. Emulators are prefixed with 'emulator-' while actual devices are a hash.

Now we have to build the Android project, which will generate an .apk file, and then we will install it onto the device.

```
ionic build android
adb -d install platforms/android/ant-build/HelloCordova-debug.apk
```

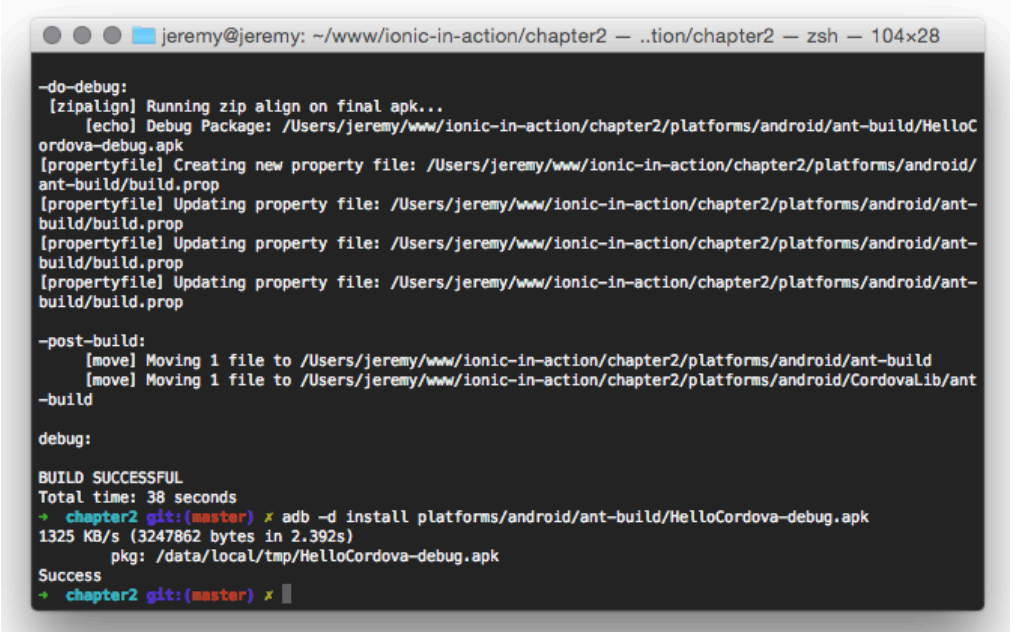


Figure 2.14 Using the Ionic build tool and Android deployment tool, the app is built and installed onto a connected Android device.

You will be able to find the app HelloWorld in the app pane. Opening it will display the same app that ran in the emulator.

2.3 Summary

This chapter covered the steps to setup your development environment and build out a sample app. While the same app was very basic, the rest of the steps were important as they will be used frequently as you begin to develop your own apps. In this chapter we covered the following concepts.

- Some software setup is required to begin developing hybrid apps.
- The command line utility for Ionic provides many features such as tools to start a project, build a project, and preview the app in a browser.
- Previewing apps in a browser is the primary environment for development and debugging.
- Emulators are great for previewing your app, and we covered how to set them up.
- You can preview the app on a connected mobile device with the proper setup.

3

What you need to know about AngularJS

In this chapter you will learn:

- How AngularJS apps are built and structured
- About underlying fundamentals of AngularJS needed to leverage the power of Ionic
- How to use controllers, filters, directives, scope, and more

AngularJS is a web application framework and its popularity has make it one of the most used JavaScript tools available today. Ionic is built on top of Angular, so it is important to have a grasp on how it works. Instead of having to build an entire web application framework for Ionic, it uses Angular and extends it with a large number of interface components and other mobile friendly features.

This chapter will walk through the core of what Angular is and cover most of the fundamentals you need to know to be effective. If you are already quite comfortable with Angular, then you can skim the chapter or jump ahead. This chapter is for those who are new to Angular or have minimal experience and need a good primer.

We will look at controllers, which are aptly named since they are designed to control your data. Then we'll discuss scope, and how it works as a glue between the controller and the user interface, which is called a view. By looking closer at views, we will see how they are built using templates and the scope to create the interactive visual experience. Along the way, we'll also look at some other features such as how to use filters to transform data, how to build and use directives to enhance regular HTML elements, and how to work with an external data source to load and save data for our applications.

This chapter will teach you about Angular by building up a basic web application. You can work through the examples or look at the complete example available on GitHub at https://github.com/ionic-in-action/chapter3. The application demo is also available to see the end result at https://ionic-in-action-chapter3.herokuapp.com/.

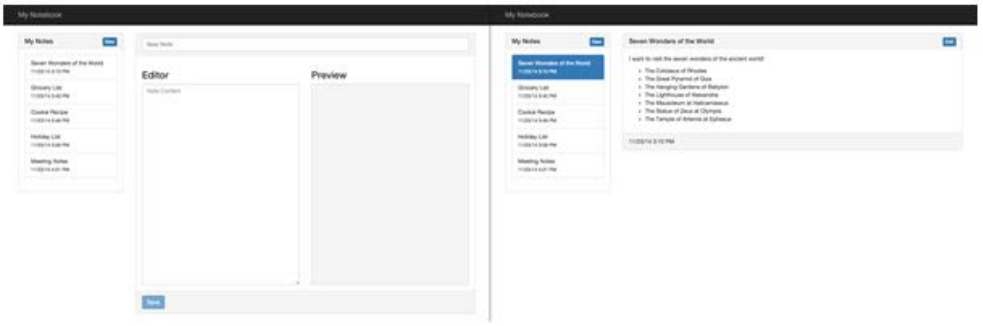


Figure 3.1 – The chapter application will have a list of notes and a way to view and edit notes.

By the end of this chapter you will have a working understanding of how Angular works. There is too much to properly cover everything Angular has to offer. Other books and material can provide you deeper Angular training, but I will cover the primary features used in this book.

Before we jump into the code, let's take a moment to talk about Angular in general and what problems it helps to solve when building web applications.

3.1 AngularJS at a glance

Before we build an application with Angular, let's take a look at the parts that make up a typical application built with Angular, starting with the views and templates used to display content and then looking at how data is loaded by controllers into the view. Figure 3.2 visualizes the way that these pieces create the list of notes in our application and demonstrates how they are connected.

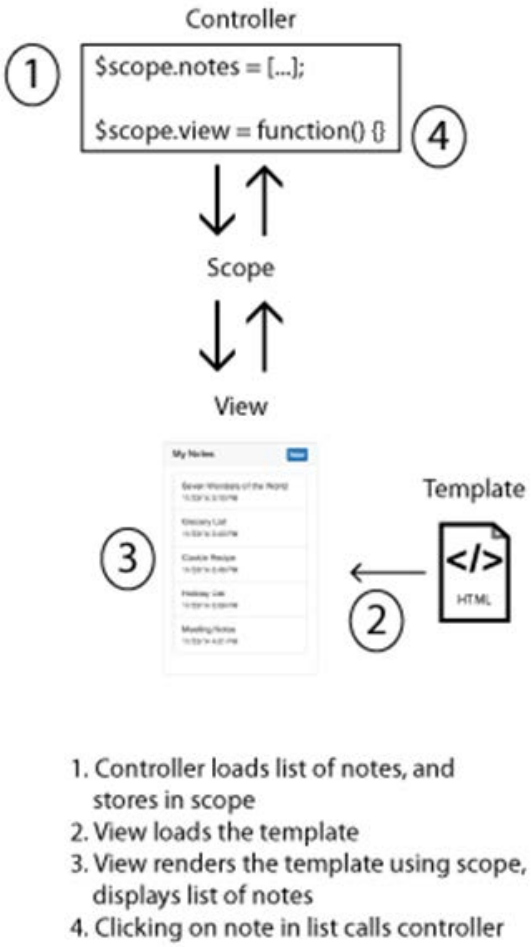


Figure 3.2 – How the list of notes is rendered from loading data in the controller to displaying in the view

3.1.1 Views and templates: Describing the content

Angular works very closely with HTML, and often you will be creating templates. A template is a block of HTML that is loaded into the application when needed. Angular enhances the HTML with new features and abilities by extending HTML's vocabulary.

A view is the use of a template to display data. A view will always have a template (which is the HTML markup) but also includes the data used. A view will transform the template into the final visual experience for the user, which means it will modify the template based on the data. A snippet of the template from our example in figure 3.2 is here.

```
<ul class="list-group">
  <li class="list-group-item"
```

```
      ng-repeat="note in notes" #1
      ng-click="view($index)"  #2
      ng-class="{active: note.id == content.id}">{{note.title}}<br /> #3
    <small>{{note.date | date:'short'}}</small></li> #4
  </ul>
```

#1 ngRepeat lets us create an LI for each note in the notes array
#2 ngClick will call the view() method when a LI is clicked
#3 ngClass allows us to conditionally show a class, in this case

This sample template shows just one LI element inside of an UL, but has several attributes that are called Angular directives. A directive modifies the behavior of the element it is placed on. In this case, `ngRepeat` will loop over a JavaScript object or array and create an LI element for each. The `ngClick` is like the JavaScript `onClick` event handler, and will call a function called view when clicked. When this template is rendered, it will create a new list item element for each note in the `notes` array.

The double curly braces indicate that some data is to be displayed here. This concept is called data binding, and the syntax is known as an expression. Anything between the braces is the expression, which is evaluated by Angular against the current model data. Therefore the content of `note.title` will be injected into the `li` element where the double curly braces wrap it.

The template is the HTML with any additional directives or expressions. The view then takes data and renders the template for final display based on the values in the data. Assuming the `notes` array has 5 notes, the `UL` element will contain 5 list item elements, like you see in the screenshot of the view in figure 3.2.

Angular comes with a large number of directives, and they all start with `ng-`. Some help to modify the display (`ngShow`, `ngClass`), while others are used with forms (`ngModel`, `ngForm`), and yet others are useful to listening to events such as clicks (`ngClick`, `ngMouseover`). Angular also has some directives that sit on top of other directives, such as inputs, textareas, and anchor tags, and provide additional features that HTML doesn't have by default. We will use more directives in our example later, but the full list is on available via the Angular documentation.

Why are directives named both ngApp and ng-app?

When people write about Angular, they can refer to directives as `ngApp` or `ng-app`. In reality they are talking about the same thing, but there is a reason why both exist.

When you see `ngApp` or `ngClass`, this is the JavaScript version of the name. JavaScript syntax rules do not allow a hyphen to be used in a variable name, so instead the convention is to use camel case but starting with a lower case. The documentation uses this style and is used in this book as well.

When you see `ng-app` or `ng-class`, this is the HTML version of the name. HTML is case insensitive and allows hyphens in tag or attribute names. The convention is to use a hyphen to increase the readability of the directive in the markup.

Now let's discuss next how data is connected to the view and displayed.

3.1.2 Controllers and scope: Managing data and logic

Controllers are functions that are attached to a DOM node and are used to drive the logic of your application. A controller is essentially a function object in JavaScript that can be used to communicate with the scope and respond to events.

The scope is like a central authority between the controller and view. Think of it as the connection between what happens in the controller and in the interface, and when the scope is updated in the controller it also updates in the view. You can see a diagram of how these pieces work together in figure 3.2 where the arrows indicate both the view and controller communicate using the scope as the hub.

Let's take an isolated example of a controller that would pair with our view and template from earlier.

```
angular.module('App')
.controller('Controller', function ($scope) { #1
  $scope.notes = [ #2
    { id: 1, title: 'Note 1', date: new Date() }, #2
    { id: 2, title: 'Note 2', date: new Date() } #2
  ]; #2
  $scope.view = function (index) { #3
    $scope.content = $scope.notes[index]; #3
  }; #3
});
```

#1 Declare the controller and use the $scope service to access scope
#2 Create an array of note objects, which the ngRepeat will display
#3 Add a method, which the ngClick on the view can call

This controller would be able to drive the template we saw before to display two notes in the list. The controller and view both can access the scope values, so when we set the notes array on the scope the view is able to also access it. The view method is a way for our application to handle tasks, in this case the task of viewing a note (which we will add the complete example later in this chapter).

Everything in this controller is isolated from other controllers. This is important because it limits the visibility of code and variables. A common challenge for new Angular developers is to accidentally put things into different scopes and have trouble accessing values in a different scope, which is not possible by default.

Angular scope is also hierarchical. Scopes can be nested, just like the DOM. In fact, a scope is reflective of the DOM structure on the page. A scope can be attached so that it is only visible to an HTML element and its children, just like how a CSS class can be used to target styles of the element it is placed on or its children.

Hierarchy becomes particularly important if you want to communicate between scopes, because a child scope can look upward to its parents (just like how JavaScript has prototypical

inheritance, if you are familiar with that concept). Some directives in Angular create child scopes for you, which can cause some confusion about which scope is where. If you look for a value on a child scope and it doesn't exist, it will actually check each parent scope for that value until it either finds the value or runs out of parent scopes to check.

There is a root scope, which is the first scope created by an Angular application to which all other scopes are attached. This means anything you put on the root scope is available to any scope, which might sound helpful but is not advised. You want to keep your scopes clean and focused instead of piling everything into the root scope. JavaScript in general has this type of a problem, where often applications use the global scope to store variables. Imagine you have a value called `id`, if you ever placed a value `id` in a child scope it would conflict and cause you to loose access to the root scope value. This becomes a problem as we incorporate more code, because the more people who work on an application or the more external tools we incorporate the more difficult it is to be aware of all that happens in the application to avoid these kinds of naming collisions.

Controllers are not for everything

There are a few things you should not do with a controller, because it can make your code hard to maintain and test later. The primary offense is doing DOM manipulation in a controller. Imagine you were building a slideshow, the controller should not handle the task of changing the DOM and CSS for the slideshow features, that would be best placed in a custom directive.

You should also avoid using the controller to format or filter data, instead use form controls and filters.

3.1.3 Two-way data binding: Sharing between controller and view

One of the most powerful features about Angular is two-way data binding. You've seen how a view binds data into the template, but it also works the opposite direction. The view can change values on the scope from the view, which are immediately updated in the scope and reflected in the controller. This is particularly helpful with forms, such as when a user types into a text input and the value of the scope changes as the user types. You don't have to do anything special to enable two-way data binding, it happens automatically for you.

In our application you will see two-way data binding happening when we set up the editor. As you type into the editor box, the contents will be previewed on the right. We'll also see this in action in most of the Ionic apps we build.

That sums up the key concepts of Angular that help give you the basic background needed to get started. Let's see how these concepts really work in our chapter project.

3.2 Setting up for the chapter project

In this chapter, we will build an Angular app together from a base HTML page. I've already done some work by creating the design and markup for the foundation so we can focus just on the features Angular builds.

This application is a simple note storing application, where you can load and modify a list of simple notes. The features for the app include the following.

- Store notes in JSON file
- View, create, edit, and delete notes
- Use markdown formatting in notes
- Editor and preview of markdown side by side

The application has been setup with the base HTML and CSS required. It also contains a simple RESTful server written with NodeJS to allow us to manage our list of notes, and thisis provided so we can focus on Angular and not the API. We will focus solely on how to add Angular into this base and cover the major features of Angular along the way.

3.2.1 Getting the project files

During this chapter, you will be able to follow along using git tags to checkout specific versions of the code. You can also follow along by writing the code yourself from the book examples. Even if you aren't familiar with git, you can run the commands to follow along, or use the second option and download the base application files and code along.

Using git, you can get started with this chapter base by cloning the chapter3 repository, and then checking out the step1 tag as shown below.

```
$ git clone https://github.com/ionic-in-action/chapter3.git
$ cd chapter3
$ git checkout step1
```

If you don't want to use git, you can also download and extract the base application files for starting at https://github.com/ionic-in-action/chapter3/archive/step1.zip.

You can use the same step to get the code for every step, just change the number of the tag in step1 to the current number.

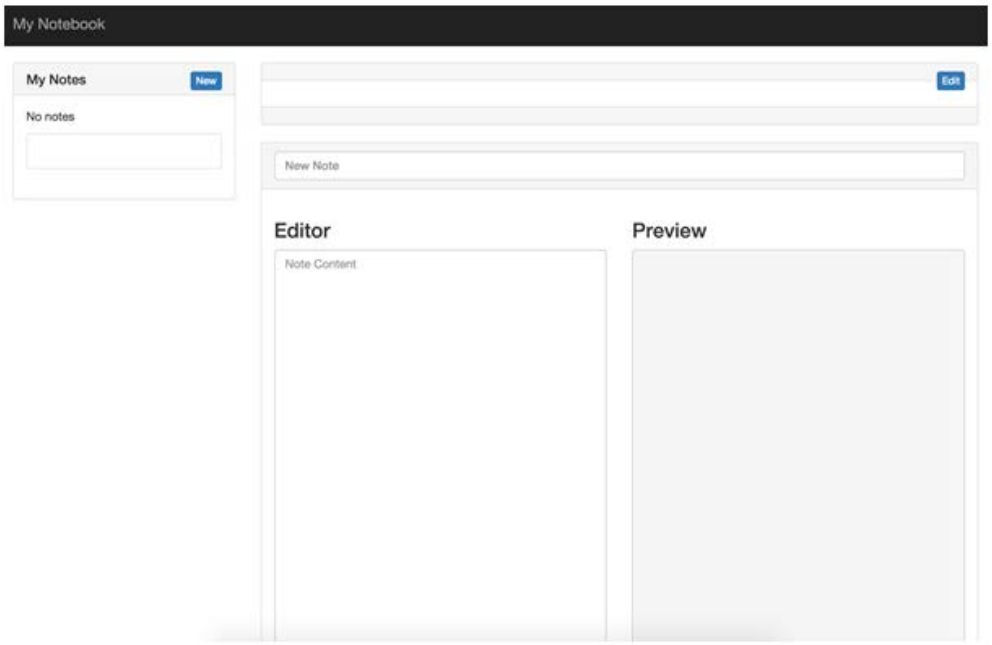


Figure 3.3 – The base HTML template for our application, which currently is not interactive.

3.2.2 Starting the development server

Now that you've got the project files downloaded onto your computer, we need to get our development server setup. For our web application to work properly, we need to have a server running like it would in a production environment.

In the server.js file that you see in the project is a simple RESTful server based on the popular Express.js framework. The primary reason for this is that we want to have a way to keep track of our notes over time, and the RESTful API will allow our application to read, create, edit, and delete notes in our list. The server also takes care of loading files into the browser over HTTP, which is essentially what the `ionic serve` command does for our Ionic apps.

I've commented the server with some notes, and if you are interested you can dig into it more. I will not be covering it in detail here, but it is important to know a few things about it.

- The server runs on port 3000, which means you have to visit http://localhost:3000 to view the web application.
- The server accepts incoming requests, and depending on the URL and HTTP method used it will modify the list of notes.
- The server uses a JSON file (data/notes.json) as a database to keep it simple. In real world applications you would use a more robust database solution.

The server will not run until we've downloaded some NodeJS packages that it requires. This is easy to do by running the following command to use NPM (Node Package Manager) to install the required files. First navigate to the directory in the terminal and then run:

```
$ npm install
```

This will take a few moments, as NPM looks at the list of dependencies (found in `package.json`) and downloads them from GitHub. When it is completed, it will return you to a prompt with a list of the packages it installed.

Now we can start our server, and it must continue to run in the command line. It will start our server and it will listen for requests on port 3000.

```
$ node server
```

At this point, you can visit http://localhost:3000 in your browser and you should see a base template layout like you see in figure 3.3. We will be modifying the HTML and adding JavaScript to bring this base layout to life as a note taking application.

3.3 Basics for an Angular app

The fundamentals of Angular start with creating an Angular application in our JavaScript and then adding a reference to it in our HTM. Angular works closely with the page DOM, so you actually restrict an Angular application to a DOM element and its children. In this case this will be the `<html>` element, so that Angular has access to the entire page. Ionic often uses this on the `<body>` element. In figure 3.4 you will see the same content in the browser as before, but it will have Angular installed and ready to use.

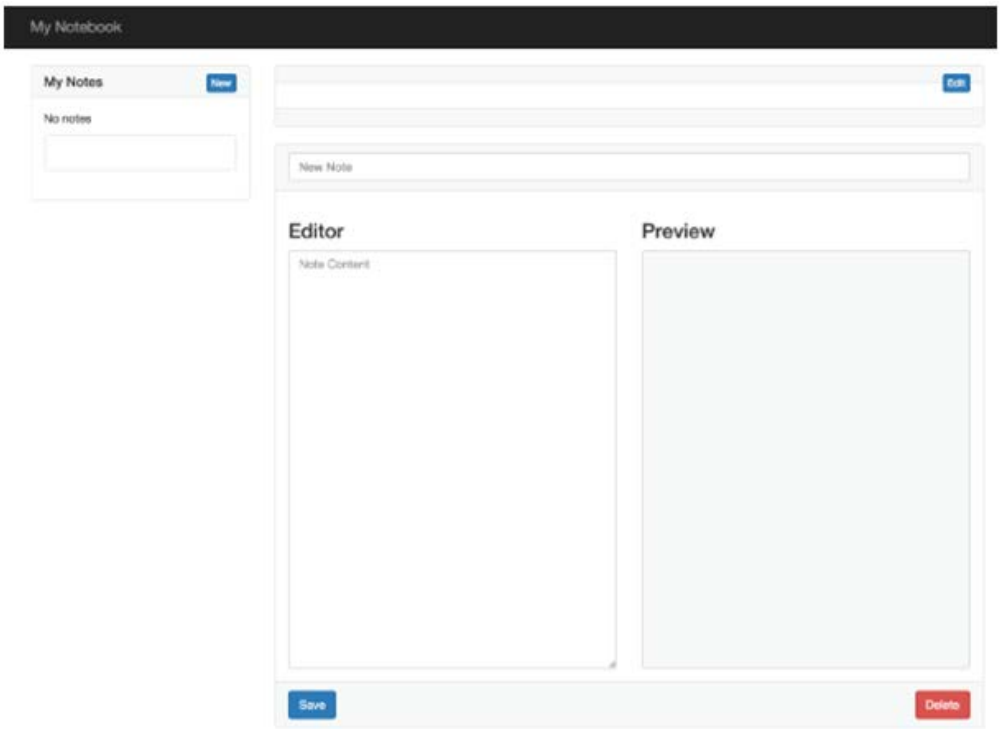


Figure 3.4 – When Angular is added to the page it still appears the same. We have to tell Angular to do some things before the content will change.

You can get the code for this step if you have cloned the repository from GitHub by running the following in your command line. This will reset any changes you've made and set the code to the step2 tag.

```
$ git checkout -f step2
```

In order to create an Angular application, we use the `ngApp` directive on an element and declare the name of the application. Open the index.html file and add the `ngApp` directive as you see here.

```
<html lang="en" ng-app="App">
```

At this point, we are attaching an Angular application called App to the root HTML element. This gives our Angular app access to the entire DOM, but you could have attached it also to the body tag. I recommend putting it on the `<html>` or `<body>` element.

We haven't yet declared this App yet in JavaScript, so let's do that now. Angular has a module system, which is a mechanism to help encapsulate program code into individual pieces.

When you declare a new module, you provide a name and then an array with a list of dependencies (which this chapter has none). Ionic itself is an Angular module, which you will declare as a dependency in other chapters. Angular modules are declared in the following way. Create a new file in js/app.js and add the following line to it.

```
angular.module('App', []);
```

Lastly, we need to add a script tag to the index.html file to load our Angular module. In the index.html, right before the closing </body> tag, add a new script tag as shown below. You need to make sure this is after the Angular library, JavaScript files are loaded and executed in the order they are declared on the page.

```
<script src="js/app.js"></script>
```

We've just declared and attached the most basic of Angular applications to our page. The `angular.module()` method creates our module, and attached it to the DOM with `ngApp`. This is the most basic Angular application, and in fact it does nothing yet. All Angular apps are declared in this basic way.

3.4 Controllers: For controlling data and business logic

Let's get some of our business logic wired up into the application. Here we are going to add a controller to manage business logic that controls the various parts of the application. This step will not change the way the application appears in the browser just yet, since a controller is about managing data and not the visual aspect of the application. However, we need the controller in place before we can start to manage the visual elements.

The result of adding a controller will give the controller a particular region on the page which is has access to, as you can see in figure 3.5. For example, we need to be able to manage how we load our data and attach it to the scope. You can reset the project to step3 if you are using git.

```
$ git checkout -f step3
```

In listing 3.x we declare a basic controller. We first have to reference the App module and then declare a controller with the controller method. We pass the name of the controller and the function that contains the controller logic. Create a new file in `js/editor.ctrl.js` and add the following code from listing 3.1.

Listing 3.1 – Editor controller (js/editor.ctrl.js)

```
angular.module('App') #1
.controller('EditorCtrl', function ($scope) { #2
  $scope.state = { #3
    editing: false
  };
});
```

#1 Reference our App module to attach the controller to this module
#2 Declare a controller with the name EditorCtrl, and pass a function that has dependencies listed

#3 Create a model value and store it on the $scope

This is a very simple controller that currently only creates a simple model called state. We've injected the $scope service so we can set the state property. Remember, values on $scope are also available for the view.

Services starting with $

You'll notice that Angular services start with a $ symbol, and the same will hold true for Ionic's services. When you see a service beginning with $, it is a convention to designate it as part of the core of Angular or Ionic.

Services that we create ourselves will never be prefixed, but I will capitalize them. There is no requirement for how you name services, but the core follows the convention of prefixing with $ and I will always capitalize services in the examples.

Now we need to add the file to our index.html file to include it in our application. Add a script tag to the bottom of the HTML right before the closing body element.

```
<script src="js/editor.ctrl.js"></script>
```

The last step is to attach the controller to the DOM. This will create a new child scope for this controller to use. This is done by using a special HTML attribute, called an Angular directive, to declare where the controller should be attached. In this case, we will want to attach it to the div with a class of container.

```
<div class="container" ng-controller="EditorCtrl">
```

Here we use the `ngController` directive and declare the name of the controller we've created in our JavaScript file. This will attach the controller to the DOM and make the controller able to manage anything inside of this element. You can see in figure 3.5 below where the controller's scope is available, which is most of the page except for the top title bar.

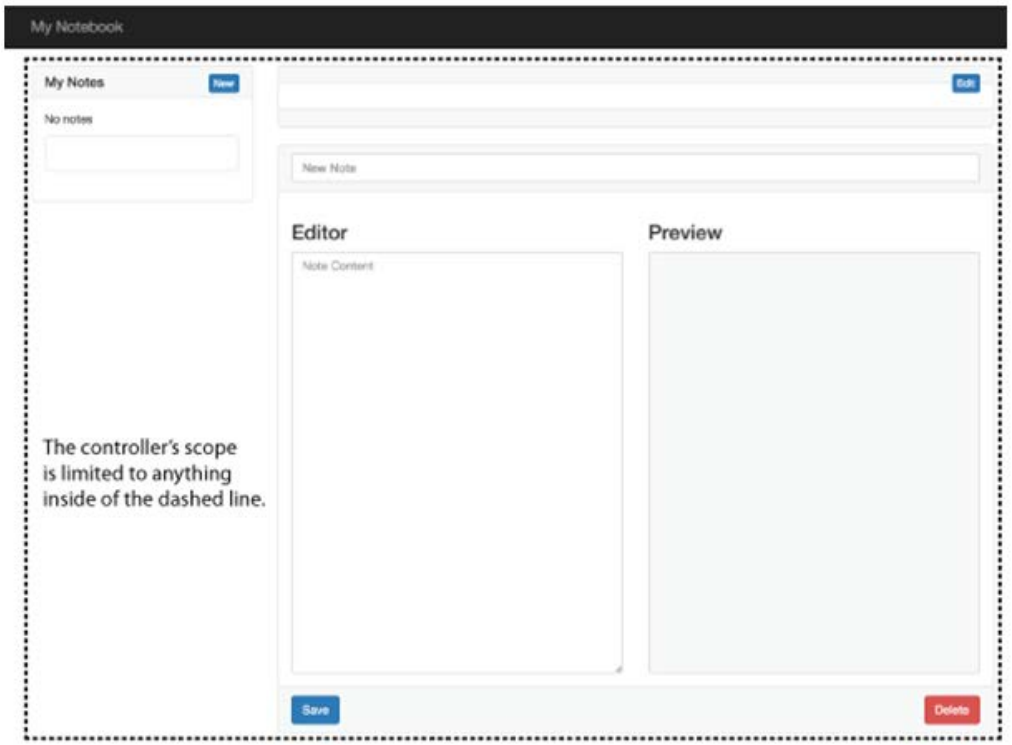


Figure 3.5 – The controller's scope can only apply to markup inside of the dashed line. The header is outside of the scope because of where we have attached the controller.

More about the server in this project

The included server does two things, it serves the static files for our application and has a RESTful API. The topic of building a RESTful API is beyond the scope of this book, but I wanted to take a moment to go over the basics of this implementation. This server runs using NodeJS, which you have already installed. NodeJS allows us to do some pretty interesting things such as work with the computer's file system and respond to HTTP requests.

NodeJS has modules that can be included in a program so you can reuse features. In this case, I have used a very popular NodeJS module called Express. Express has a lot of the built in features for building an HTTP server. I also use the file system module to maintain a list of notes in a JSON file. These are things you cannot do with JavaScript in a browser, but NodeJS makes possible.

You can look at the server.js file inside of this project to see the server code. This is a fully featured server, and it is impressive how easy it is to create with NodeJS. You can learn more about Express at http://www.expressjs.com.

3.5 Loading data: Using the controller to load and display data in the view

Let's start loading our data and getting it to display in our application. On the left we want to show a list of the notes that have already been created. I've seeded this project with a few notes to get us started. Since we've already created our controller, we can update the controller to load data into our app. To do this, we will use the Angular `$http` service which allows us to make HTTP requests to load data from our NodeJS server. Figure 3.6 shows where the application will display the list of notes. You can reset your project to step4 if you are using git.

```
$ git checkout -f step4
```

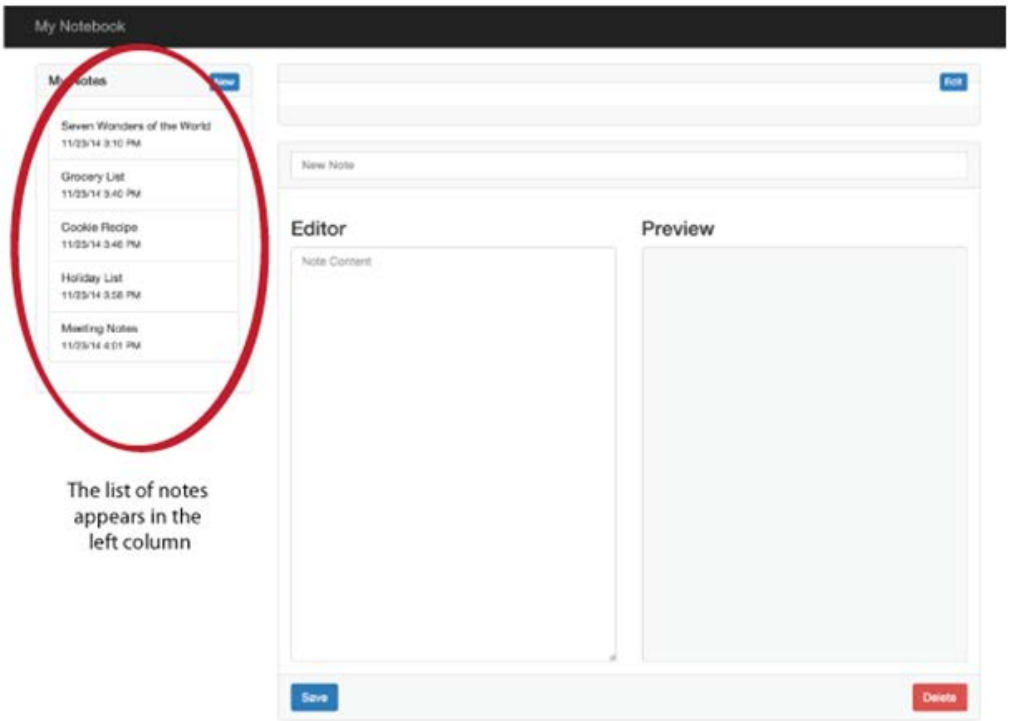


Figure 3.6 – The data will be loaded and then displayed in the list on the left, showing the 5 default notes.

Let's modify the controller to add the HTTP request to our notes service, and assign the resulting data to our scope. Open up the js/editor.ctrl.js file and update it to the following in listing 3.2.

Listing 3.2 – Editor controller loading notes from service (js/editor.ctrl.js)

```
angular.module('App')
.controller('EditorCtrl', function ($scope, $http) { #1
  $scope.editing = true;

  $http.get('/notes').success(function (data) { #2
    $scope.notes = data; #3
  }).error(function (err) { #4
    $scope.error = 'Could not load notes'; #4
  }); #4
});
```

#1 Inject the $http service into our controller
#2 Use $http.get to load the notes, on success handle the data returned
#3 Attaching the returned data from http to the $scope
#4 Handle the error, store an error

This now will make an HTTP request to http://localhost:3000/notes to load the default list of notes from the `data/notes.json` file as soon as the controller loads. You can inspect the network requests in your browser inspector tools to see that the request returns the array of notes. Angular takes care of automatically parsing the JSON into a JavaScript object, as long as the `Content-Type` header from the API is set for `application/json`. This makes it easy to load JSON data without having to handle the parsing yourself.

In your controller function, you can declare any number of parameters for the function and Angular will try to locate a service by that name and inject it into the controller. For example, we are able to inject the $http service into our controller (#1) and then use it to load data (#2). This is called dependency injection, and is a powerful feature of Angular to be able to make services available for your controllers to use. Angular services are not global and cannot be used without first being injected.

I like to think of dependency injection like a waiter at a restaurant. A waiter comes to your table and takes your order. He takes that to the kitchen, works with the kitchen staff to prepare the dish, and returns with the meal to your table. The DI system takes your order for which services you need, does any work to set them up, and returns the services to your function for you to use. We are able to inject the default Angular services, or any of the services that we create ourselves.

In our code there are two methods chained to the `$http.get()` method. The code inside of the `success()` function will run when the data has loaded while code inside of the `error()` function will run if there was a problem getting the data (such as the HTTP request failed because the server was down).

Angular and asynchronous methods

JavaScript is single threaded, which means it can only execute one task at a time. Certain tasks, like loading data from a server, can take a reasonably long time. In synchronous programming this would block other tasks code from running and until it was finished, likely causing the interface to freeze

during this time. Fortunately, JavaScript does not do this. JavaScript supports many asynchronous tasks, which solve this problem.

When JavaScript runs an asynchronous task it begins with the first part of the task, and then set it aside to continue running other tasks. When the asynchronous task finishes, it alerts JavaScript it is finished and the rest of the task is queued to execute. This frees up JavaScript to continue processing tasks. HTTP requests in JavaScript (also called AJAX or XHR requests) are one example of an asynchronous function, since there is a lot of time spent waiting for the server to respond.

There are two primary ways to handle asynchronous functions, callbacks and promises. Angular uses promises for asynchronous method calls, but both may be used depending on the structure of the application or modules you are using.

To get more details about promises with Angular, I recommend looking at his blog post from Xebia's blog http://blog.xebia.com/2014/02/23/promises-and-design-patterns-in-angularjs/.

Now we can't yet see any of the data on our screen, so let's update our template file to show the list of notes in the left column. This will use template binding and several Angular directives to manage the display of this data from the `$scope`. Open up the index.html file and look for the markup below in listing 3.3 and add the bolded parts into the template.

Listing 3.3 – Notes list template (index.html)

```
<div class="col-sm-3">
  <div class="panel panel-default">
    <div class="panel-heading">
      <h3 class="panel-title"><button class="btn btn-primary btn-xs pull-
       right">New</button> My Notes</h3>
    </div>
    <div class="panel-body">
      <p ng-if="!notes.length">No notes</p> #1
      <ul class="list-group">
        <li class="list-group-item" ng-repeat="note in notes">{{note.title}}<br /> #2
        <small>{{note.date | date:'short'}}</small></li> #3
      </ul>
    </div>
  </div>
</div>
```

#1 ngIf conditionally includes or removes the element from the DOM when there are notes or not
#2 ngRepeat will loop over every note and display the note title
#3 This binding shows the date, but also formats the date using the 'short' format date filter

Here our template is now displaying the list of notes once the controller has loaded them. While the list is loading or if no notes were found, the `ngRepeat` list would be empty and the `ngIf` would display the "No notes" message. The expression is evaluated every time the notes model is updated, so as soon as the notes model has at least one item in the array the expression `!notes.length` will return `false` to hide the paragraph element. This is a simple way to use Angular's directives to modify the template based on values attached to the `$scope`.

The `ngRepeat` will loop over every item in an array (or property of an object) and create an element for each item. In this case, there will be an `LI` element for each note in the array, and it will display the title and date the note was last saved.

You can explore the large number of directives that Angular provides to see all of the features they can provide. We will be using a number of them in our Ionic apps, but I will provide some detail about any new ones as they are used.

3.5.1 Filters: Convert data to display in the view

The `note.date` data binding in our template is followed by `| date:'short'`. This is called a filter, which will modify the display of the binding without changing the value on the scope. For example, here we have a Date object, and using the Angular date filter the display formats it to a human readable format while retaining the original Date object on the scope.

Filters are used in expressions by adding the pipe character and then the filter. Filters can be chained together, or in other words you can add more than one filter. For example, a filter could sort an array (using the `orderBy` filter) and then another filter could reduce the array to 10 items (using the `limitTo` filter). The expression with the filters would appear like this.

```
{{notes | orderBy:'title' | limitTo:10}}
```

Angular has a handful of filters by default, such as a currency filter to format a number as a currency value (like $100.00 for USD or €34 for Euro) based on browser settings. Filters can also be used as a service, but that is less common.

3.6 Handling click events to select a note

Now we need to be able to view these notes individually. We will want to click on a note in the list on the left and have that note appear in the right. Figure 3.7 shows how the click will select the note and then display it on the right. You can set your git repository to this step by checking out the step5 tag.

```
$ git checkout -f step5
```

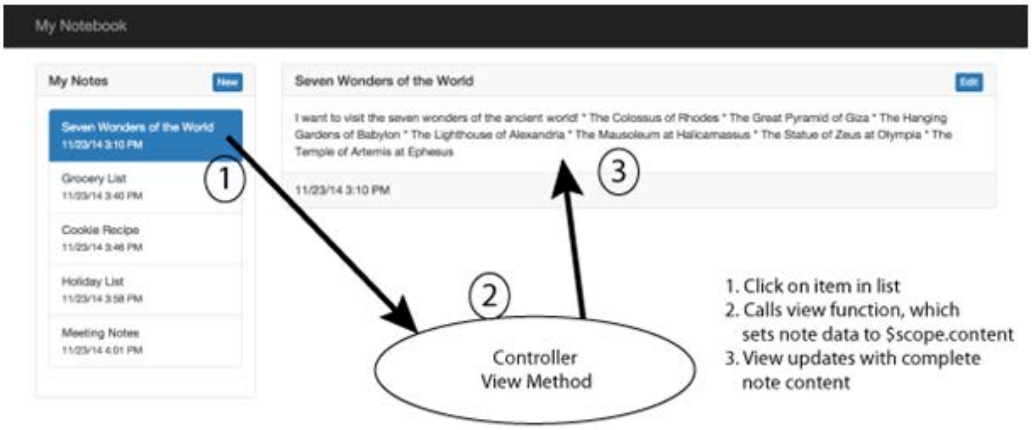


Figure 3.7 – Clicking on an item calls the controller and updates the view with the selected note

We will accomplish this by using `ngClick` to handle clicking on the item, and then assign the data from the note to a new model for display. Look at the template again and we will modify the section with the list of notes to add the click handler found in listing 3.4.

Listing 3.4 – Adding ngClick to note list (index.html)

```
<ul class="list-group">
  <li class="list-group-item" ng-repeat="note in notes" ng-click="view($index)" ng-
      class="{active: note.id == content.id}">{{note.title}}<br /> #1
  <small>{{note.date | date:'short'}}</small></li>
</ul>
```

#1 Add ngClick and call a scope method called view with the index of the clicked item. Add the ngClass which will add the 'active' class if the note currently

Each note in the list can now be clicked, and when clicked Angular will try to call the `$scope.view()` function. The `ngClass` directive is a useful way to conditionally apply a CSS class to the element. In this case the active class is used to highlight the item after we have clicked on it to view the note.

The `$index` value is passed to the view function, and it is a special variable that `ngRepeat` provides. It helps us know what the index of the array item is, and in this case the index of the note which has been clicked.

We haven't created the view function yet, so let's do that now. Open the editor controller and add the view function found in listing 3.5 into the controller function.

Listing 3.5 – View function in the editor controller (js/editor.ctrl.js)

```
$scope.view = function (index) { #1
  $scope.editing = false; #2
  $scope.content = $scope.notes[index]; #3
};
```

#1 Declare a new $scope method called view and accept the index of the clicked item
#2 Set the editing state to false, since we want to just view the item
#3 Set a new model for the content model to contain the note that was clicked

Now the click event will fire the `view()` method in our controller when the note is clicked. It sets a new content model, which contains the data from the note that was clicked, using the index value that was passed. It also sets the editing model to false, because anytime we view an item it should reset to display mode and not editing. The editing portion will be wired in a few more steps.

We are now handling the click event and setting the content model with the note data that was selected. However we aren't able to see the note yet in the view, so let's update our template to display the selected note properly.

We've created a new content model which contains our note, but we have to update the template to show the note. You can set your git repository to this step by checking out the step6 tag.

```
$ git checkout -f step6
```

We need to modify the right column of the application to display the two panels properly. So far the right shows two panels, and we need to configure so only one of the panels appears at once. The first panel is intended for when we want to view a note, and the second panel is for when we want to edit a note. We have already set a `$scope.editing` property, which we will use to determine which panel to show. Open the index.html file again and let's modify the right column content by adding the bolded items in listing 3.6.

Listing 3.6 – Modify template to view a note (index.html)

```
<div class="panel panel-default" ng-hide="editing"> #1
  <div class="panel-heading">
    <h3 class="panel-title">{{content.title}} <button class="btn btn-primary btn-xs
      pull-right">Edit</button></h3> #2
  </div>
  <div class="panel-body">{{content.content}}</div> #3
  <div class="panel-footer">{{content.date | date:'short'}}</div> #4
</div>
<form name="editor" class="panel panel-default" ng-show="editing"> #5
```

#1 ngHide will hide the top panel if the condition is true, in this case when editing is true
#2 Bind the title into the panel header
#3 Bind the content into the panel body
#4 Bind the note date and pass it through the date filter to use the short format
#5 ngShow will hide the bottom panel if the condition is false, in this case when editing is false

When you run the application now, you are able to click and view each note individually. The template will now react to the changes made in the `view()` method which set the content and editing models. When the editing model is true the editing panel will appear, otherwise the note panel will display. The `ngShow` and `ngHide` directives are useful to toggle the display of

elements like shown here. it sets editing to false and sets the content model to contain the selected note.

We've added the bindings for the note title, content, and date. The date has the date filter applied just as we did earlier. Now we need to create a new directive that will parse the note content to display it properly.

3.7 Create a directive to parse the note with Markdown

At this point we can view our existing notes, but the formatting isn't quite right in the notes. This application will support writing notes with Markdown, which is a simple way to write text that can be easily converted into HTML markup. You can learn more about Markdown at http://daringfireball.net/projects/markdown/. Figure 3.8 shows the area that will be formatted using Markdown. You can set your git workspace to this step by checking out step7.

```
$ git checkout -f step7
```

Figure 3.8 – Note content written in Markdown format will now be parsed and converted into HTML

We are going to create a simple Angular directive that can convert the plain text with Markdown syntax into HTML. To do this I'll be using the popular JavaScript markdown library called Showdown. It is already included by default in the application file.

To create our directive, I'm going to open up the app.js file. Directives are not part of a controller, so I am organizing my code so it is stored in the main app definition. In listing 3.7 you can see the directive we'll use to convert Markdown to HTML.

Listing 3.7 – Markdown to HTML directive (js/app.js)

```
angular.module('App', [])
.directive('markdown', function () { #1
  var converter = new Showdown.converter(); #2
  return { #3
    scope: { #4
      markdown: '@' #4
```

```
    }, #4
    link: function (scope, element, attrs) { #5
      scope.$watch('markdown', function () { #6
        var content = converter.makeHtml(attrs.markdown); #7
        element.html(content); #8
      });
    }
  }
});
```

#1 Declare the directive and name it markdown
#2 Create a Showdown converter to use later
#3 Directives return an object to define the directive settings
#4 Declare a custom scope, that expects a value to be assigned to a markdown attribute
#5 Declare a link function, which actually manages the conversion from markdown to HTML
#6 Use a scope watcher to update anytime the model changes
#7 Convert the markdown into the content variable
#8 Inject the converted HTML content into the element

This directive will automatically convert the Markdown to HTML anytime the content changes, which will become helpful while we are editing. The directive works by first creating a new Showdown converter service. The directive then is defined, and in this case the directive will have its own isolate scope nested inside of the controller scope. I've defined markdown as a property of the scope, and I'll demonstrate how that value gets populated in the next section.

The link function is used by Angular as part of the rendering process. It will use the `$scope.$watch` feature, which allows us to listen for when the markdown content is changed. When it detects a change, the plain text content is passed to the Showdown converter and then injected into the element as HTML. The source content in the scope will remain the plain text version, but the it will always appear as the converted HTML.

Let's put the directive into action and see how to pass it the markdown content. It will take our note content, parse it using Showdown, and inject the resulting HTML into the element. Open the index.html and modify the existing content binding like you see in listing 3.8.

Listing 3.8 – Adding markdown directive into template (index.html)

```
<div class="panel-body" markdown="{{content.content}}"></div> #1
```

#1 Use the markdown directive and pass the content of the note to parse

Notice how this is an HTML attribute, like the other directives we have used. We assign the `content.content` model to the markdown attribute in order to pass the content of the model to our directive's isolated scope. This directive is used as an attribute on the element which we want to inject the content into. The HTML is injected inside the `DIV` element, and anytime the `content.content` model is changed our scope watcher function will fire to reconvert the new content.

Directives are a very complex topic, and I have only scratched the surface of what you can do with them. There are also different ways this directive could have been built, which makes the directives feature in Angular so powerful but also somewhat difficult to grasp.

Will I need to write my own directives?

There is no reason that you would be required to write your own directives, so in short the answer is no. Directives exist for any situation where you want to modify an element in the DOM, but you can often manage the same logic using a controller. However, there are many good reasons you would want to create custom directives.

Directives are easier to test when they are written well. Directives encapsulate both the functionality (they can include a controller or link function) and the template (they can include template fragments). This makes them modular and isolated from other aspects of your code, making the tests focused on just the directive.

They are also reusable and reduce code that would have to be written in multiple places. Regardless of where in the application you wanted to reuse the directive, it would have a consistent behavior. If you put that logic into a controller and wanted to use it again in another controller, you'll either have to write the same code twice or work out how to share scopes.

You could build Angular and Ionic apps without your own directives. I recommend a beginner not focus on directives until they are comfortable with Angular in general. If you are comfortable, then a few ways to identify when something should be a custom directive is to look for cases where code is being duplicated or where the DOM is being manipulated from the controller.

At this point, the viewing of existing notes is complete. Next we want to focus on making the editor work.

3.8 Using models to manage content editing

The editor will have two primary functions, to edit existing notes or to create a new note. To begin we shall setup the editor so it can create a new note when the application first loads, or when the user clicks on the new button. Figure 3.9 shows the changes we will make in this section. You can set your git repository to the code for this section by checking out step8.

```
$ git checkout -f step8
```

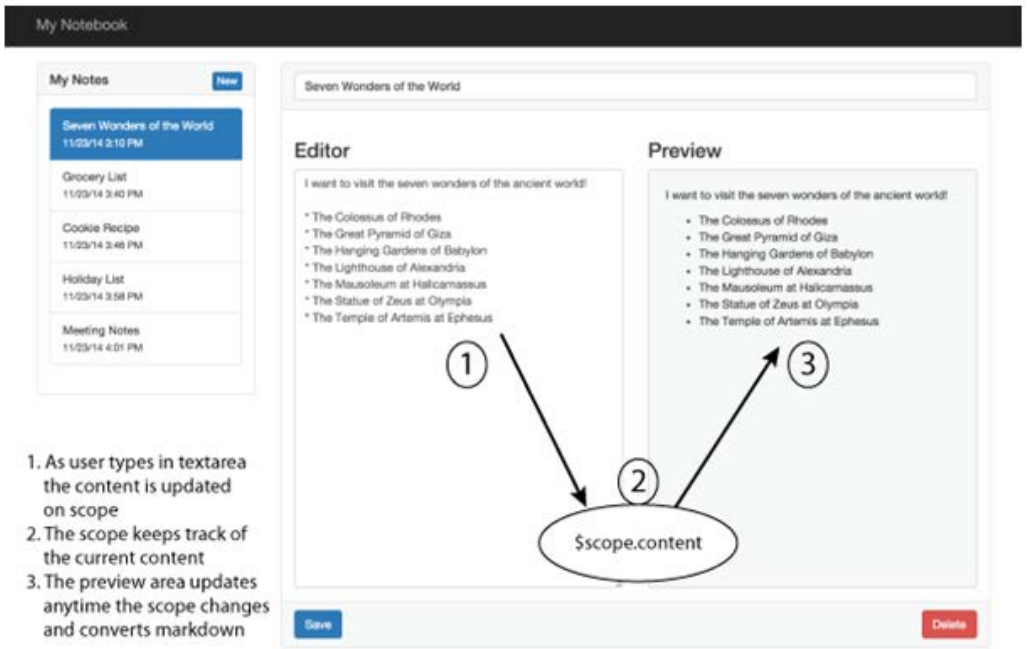


Figure 3.9 – The model is modified by the user using the textarea and the preview area is instantly updated with changes

To begin we need to add some models to our form so we can use the form controls to update the data. We also want to have the right side of the editor show a preview of the content as we type, so we shall add the markdown directive here as well.

Open up the index.html and modify the markup inside of the form to reflect the bolded contents in listing 3.9.

Listing 3.9 – Updating the editor with models (index.html)

```
<div class="panel-heading">
  <h3 class="panel-title"><input type="text" class="form-control" ng-
      model="content.title" placeholder="New Note" required /></h3> #1
</div>
<div class="panel-body">
  <div class="row">
    <div class="col-sm-6">
      <h3>Editor</h3>
      <textarea class="form-control editor" rows="10" ng-model="content.content"
       placeholder="Note Content" required></textarea> #2
    </div>
    <div class="col-sm-6">
      <h3>Preview</h3>
      <div class="preview" markdown="{{content.content}}"></div> #3
    </div>
  </div>
```

```
</div>
```

#1 Attach the title model to the input
#2 Attach the content model to the textarea
#3 Use the markdown directive to preview the content

Here we are using `ngModel` to link the model values to the input and textarea, so any changes the user types into those fields will instantly change the `content` model. Once you have these changes, you can reload the page and start to type in the editor textarea. The preview field should update immediately with the content, and if you use markdown formatting it will be converted in the preview area.

We want to allow users to click the new button to create a new note after they have already started to view existing notes, since right now the editor only appears when you first load the application. To do this we need to add a click event to the new button.

We also want to allow them to edit existing notes, so we need another button that will start editing a note after we've opened it to view. This will be done by simply changing the `editing` model, which will show the editing panel and hide the note panel.

In the index.html update the new button in listing 3.10 with the bolded code and then in listing 3.11 update the edit button with the bolded code.

Listing 3.10 – New button handler (index.html)

```
<h3 class="panel-title"><button class="btn btn-primary btn-xs pull-right" ng-
      click="create()">New</button> My Notes</h3> #1
```

#1 Add click event handler to call create method on scope

Listing 3.11 – Edit button click event (index.html)

```
<h3 class="panel-title">{{content.title}} <button class="btn btn-primary btn-xs pull-
      right" ng-click="editing = true">Edit</button> #1
```

#1 Add click handler to change to editing mode

The new button will try to call the `create()` method from the controller, which we will define next. The edit button doesn't call a method, but will actually update the value of the `editing` model to set it to `true`. You could also have written this as a function, but since we can use expressions in our template this also does the trick.

Let's define the `create()` method in our controller now, so open up the editor controller and add a new method that you see in listing 3.12.

Listing 3.12 – Create note controller method (js/editor.ctrl.js)

```
$scope.create = function () { #1
  $scope.editing = true; #2
  $scope.content = { #3
    title: '', #3
    content: '' #3
  };
};
```

#1 Create method, attach to scope so it can be called from ngClick in template
#2 Ensure the editing state is set to true
#3 Reset the content model with blank values

When the edit button is clicked, the create method will fire. It changes the `editing` state to be `true` and then resets the `content` model for a blank note. This will cause the editor to appear and a blank note to be displayed in the form, which is our new note.

3.9 Saving and deleting a note

Now we are able to create or edit existing notes, but we aren't able to save them yet. We need to add a save method to our controller and have the save button call it. However, we also only want to save the item if the note is valid, which means it needs both a title and some content. Figure 3.10 shows the save and delete buttons when you are editing an item, and how they call the controller methods to handle the click event. You can set your git repository to the code in this section by checking out step9.

```
$ git checkout -f step9
```

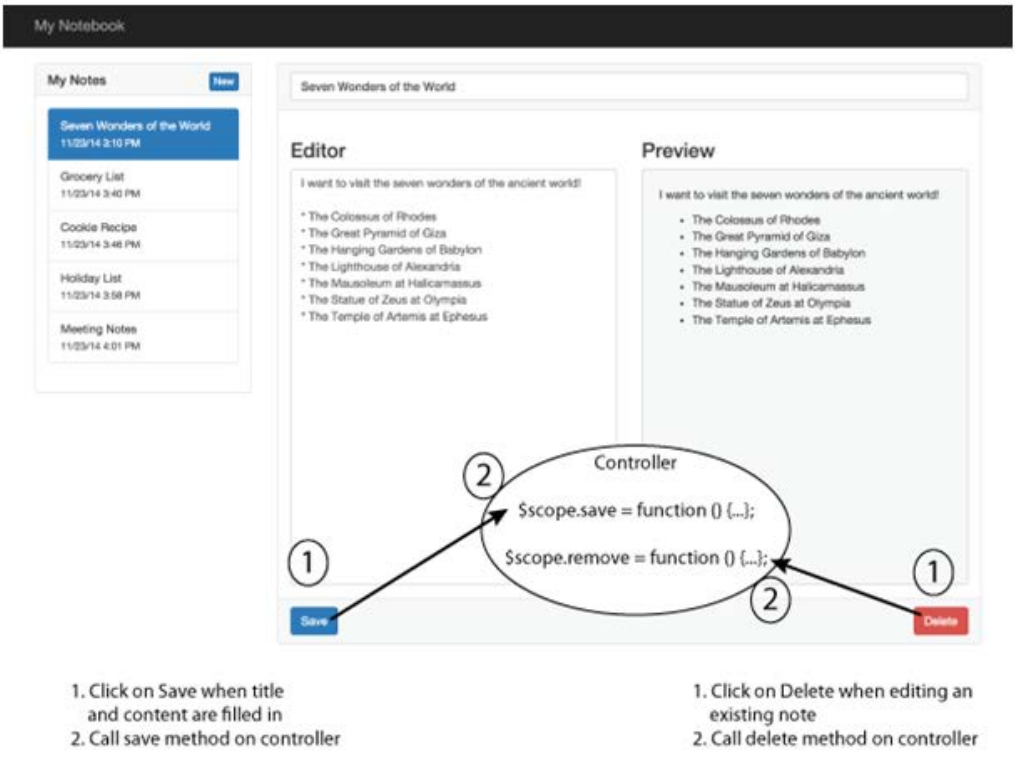


Figure 3.10 – The editor save and delete buttons call a method stored on the controller which handles the click event.

Saving a note requires using the `$http` service again to send the note to the service. Our service uses the POST method to update a note and the PUT method to create a new note. `$http.post()` and `$http.put()` both accept a second parameter, which is the data to be sent to the service. Otherwise, the syntax is the same as `$http.get()`.

Before we can save a note, we have to determine if the note is new or already exist. To do this, we look for an `id` on the content. New notes are not given an `id` until they have been saved, so if it exists we need to update. Once we know if the note is new or existing, we can call the correct service endpoint.

Open up the editor controller and add the `save()` method from listing 3.13.

Listing 3.13 – Save controller method for saving notes to service (js/editor.ctrl.js)

```
$scope.save = function () { #1
  $scope.content.date = new Date(); #2

  if ($scope.content.id) { #3
    $http.post('/notes/' + $scope.content.id, $scope.content).success(function (data)
      { #4
      $scope.editing = false; #4
    }); #4
  } else {
    $scope.content.id = Date.now(); #5
    $http.put('/notes', $scope.content).success(function (data) { #6
      $scope.notes.push($scope.content); #6
      $scope.editing = false; #6
    }); #6
  }
};
```

#1 Attach save method to the scope
#2 Set a date value of the last edited date for this note
#3 Check if this note has an id so we can either update an existing note or create the note if its new
#4 Send a post request to the notes API to update the note, and disable edit mode when completed
#5 Since this is a new note, give it a unique id based on current timestamp
#6 Send a put request to the notes API to create a new note, and then add the note to the notes list before disabling edit mode

The `save()` method starts by updating the date value with the current timestamp, since we want to store the time it was last saved. It then sends either a PUT or POST request, depending on if the note is new or existing by looking if an `id` exists. When the request has completed, both types of requests will disable the editing mode to view the saved note. If the note is new, then it is also given an `id` and then added to the `notes` array in our controller. This is important to keep both the application and the service layer in sync with one another, otherwise the new note would be stored into the service layer but not shown on the left notes list.

3.9.1 Using Angular forms for validation

Before we save an item, we'll use Angular's built in form features help us validate our form and disable the save button if it is invalid. Angular extends the default form features you know

about in HTML with a large set of features, and one particularly useful feature is automatic validation.

Notice on the form controls that we have the `required` attribute. Angular will look for that and automatically set some values on our scope to track if the forms are valid. In this case, we require that both a title and content are set for a note, so if either is blank the entire form is invalid.

Angular uses the normal HTML form element or the `ngForm` attribute to enhance forms. In this case we are using a normal form element and have given it a name of `editor`. The form then adds a new property by the same name to the scope, and has a number of values such as `$valid`, `$invalid`, `$dirty`, or `$pristine`. These values can help you understand if an input is valid or has been modified.

We will use the validation to disable the save button until the form is valid, and add a click event to the save button. In the index.html file locate the save button and add the bolded directives to the element as shown in listing 3.14.

Listing 3.14 – Adding validation and click handler for save button (index.html)

```
<button class="btn btn-primary" ng-click="save()" ng-
      disabled="editor.$invalid">Save</button> #1
```

#1 ngClick calls the save button, ngDisabled checks the Angular form for valid state before allowing the button to be clicked

The `ngClick` should be familiar by now. It will call the save method in the controller and take care of saving the note. However, it will not fire while `editor.$invalid` is true. The ngDisabled directive looks at the form validation, and disables while both form controls are empty. Angular is aware of validation attributes like required and when an `ngModel` is attached to a form field, the form can provide automatic validation.

The last feature we want to create is a way to delete a note. The delete button shows only when you've selected an existing note to edit. To get the changes in your git repository for this section, checkout step10.

```
$ git checkout -f step10
```

We will first add the `remove()` method, which will handle calling the service to delete the note and then remove it from the application. Using the code in listing 3.15 you can update the editor controller.

Listing 3.15 – Method to delete a note (js/editor.ctrl.js)

```
$scope.remove = function () { #1
  $http.delete('/notes/' + $scope.content.id).success(function (data) { #2
    var found = -1; #3
    angular.forEach($scope.notes, function (note, index) { #3
      if (note.id === $scope.content.id) { #3
        found = index; #3
      } #3
    }); #3
```

```
    if (found >= 0) { #4
      $scope.notes.splice(found, 1); #4
    } #4
    $scope.content = { #5
      title: '', #5
      content: '' #5
    }; #5
  });
};
```

#1 Declare the remove method
#2 Make the delete request to the notes API
#3 Loop through the notes to find the index of the deleted note
#4 If the note was found, remove it from the notes list in the Angular app
#5 Reset the content model for a new note

The `remove()` method sends a DELETE request to the notes service based on the `id` of the note, and then when it has returned it will remove the item from the `notes` array in the controller. To delete the note from the `notes` array on the scope, we loop over every note looking to see if the deleted id matches a note, and only if found does it splice (remove) that item from the array. It also resets the `content` model so it is ready for a new note.

The last change is to add the `ngClick` to the delete button to call the `remove()` function. We also will use an `ngIf` to conditionally show the delete button only when we are editing an existing note, because you should not be able to delete a new note that has not been saved. Listing 3.16 shows the Angular directives used on the delete button, which are bolded.

Listing 3.16 – Update the delete button (index.html)

```
<button class="btn btn-danger pull-right" ng-click="remove()" ng-
      if="content.id">Delete</button> #1
```

#1 ngClick to call the remove method, and ngIf only shows the button if the note has an id

The button will now display only when editing an existing note, and call the `remove()` method.

This now completes our Angular notebook application. In a whirlwind tour of Angular through building this application, we've scratched the surface for many of the core pieces of Angular. As we move from just Angular to building Ionic apps, you'll continue to see these features used. There are more concepts that are not covered here, because it takes far more pages than I can devote here.

3.10 Continuing with Angular

Learning Angular is important to building Ionic apps, so if this is your first look at Angular I want to encourage you to spend time with it. I believe the best way to learn Angular is to interact and build with it.

There are many opinions about how to best work with Angular. While there are certainly some best practices that have been discovered, there are also many opinions. Take care to not be overwhelmed by the vast number of posts online that discuss the "right" way to build

Angular applications. They likely have good points, but also may have opinions that don't fit your programming style or needs.

You can continue learning about Angular with the Angular in Action (http://manning.com/bford/) or Angular in Depth (http://manning.com/aden/) books from Manning. The AngularJS website (https://angularjs.org) is the primary resource for documentation, and also includes a good getting started guide. There are also many good recorded talks on YouTube that range from beginning with Angular to very specific, advanced topics (https://www.youtube.com/user/angularjs).

Even while you could create a fairly good Ionic app without knowing the inner details of Angular, you're ability to develop a great Ionic app increases as you learn and expand your Angular skills.

3.11 Chapter challenges

You've seen the fundamentals of Angular in action in this chapter, but I have a few challenges for you to dig into to improve your understanding.

- **Show errors.** Notice we set the value of `$scope.error` in the `$http.get()` method, but never did anything with it. Modify the template so that an error message will show when the value of `$scope.error` is set.
- **Handle other $http errors.** The `$http` methods all can handle error situations, and in our example only the `get()` method is setup to do this. Improve the example by adding error handling for the other methods as well.
- **Use ngResource.** Instead of `$http`, you could use the `ngResource` module which is an abstraction for interacting with a RESTful API that makes it easier to create services based on `$http`. You will need to include the module by adding the files to the application (you can use bower or download from Angular's site) and work out how to include another module in your application.

3.12 Summary

In this chapter we've covered many aspects of Angular through our example notebook application. The major things we covered are:

- Angular extends HTML with additional features that are made available through the many directives it provides or that you can create yourself.
- Templates are HTML and may contain Angular directives or expressions. These templates are converted into a view that users interact with.
- Controllers are used to hold the business logic of your application. They are functions that can have any number of serves injected into them using Angular's dependency injection system.
- Scope is the glue that holds the controller and views together, and powers the two-way data binding in Angular. When data changes in the view or controller, the data is automatically synced to the other.

- Filters are ways to transform data in a template without modifying the source model stored in the scope.
- Directives are powerful and you can create your own when you are comfortable with Angular. They are not required, but should be used when appropriate.

4

Build an app with navigation

In this chapter you will learn how to:

- Manage the user state and navigation
- Display lists and use cards to organize content
- Use icons and use content containers
- Load data and show a loading screen
- Use a slideshow component as an app intro

In this chapter we'll create a fully functional mobile app for a fictitious resort in Hawaii. The core feature of this app is how we'll manage user's navigation through various parts of the application. I'll also introduce you to some of the Ionic components during this chapter, such as loaders, content containers, and a slideshow.

This chapter is laid out to provide a walk through of the process to build the complete app. The complete example is available on GitHub if you'd like to take a look at the whole app first. You can find it at https://github.com/ionic-in-action/chapter4.

A vital part of any mobile app is to manage the user's ability to navigate through the app. I will begin by setting up the necessary foundation for the application navigation. Then I will build from that foundation and add new views that introduce additional Ionic user interface components. All of the components will work together to provide a mobile app that shows the current weather details, a visitor's reservation details, and upcoming events at the resort. It will also include an introductory tour of the app using a simple slider, which you may have seen in other mobile apps. At the end of the chapter I will give you a list of several suggested challenges to improve the app and practice what you've learned.

In figure 4.1, I have outlined the basic app flow. This provides a general idea of what the user will be able to do and where they can navigate. There are notes about the basic features for each of the views. Some type of wireframing like you see in figure 4.1 is helpful in the planning stages of any mobile app.

Figure 4.1 – Resort app wireframe showing the views and user flow through the app.

Now let's get to building this app!

4.1 Setup chapter project

During this chapter we'll be building out the app by first using the Ionic CLI tool to generate a new project. Open the command line, navigate to the directory you'd like to store your project,

and create the project with the command below. This will setup the project for you and you can go back to chapter 2 to review the process in more detail. It will download the starter project from the Ionic in Action GitHub account. When the process has completed, you will have a new blank project to start with.

```
$ ionic start chapter4 https://github.com/ionic-in-action/starter
```

Before we build our resort app take a look at figure 4.2. It shows the various places in the app that a user will be able to go. We'll build out each piece individually but here you can see them all together.

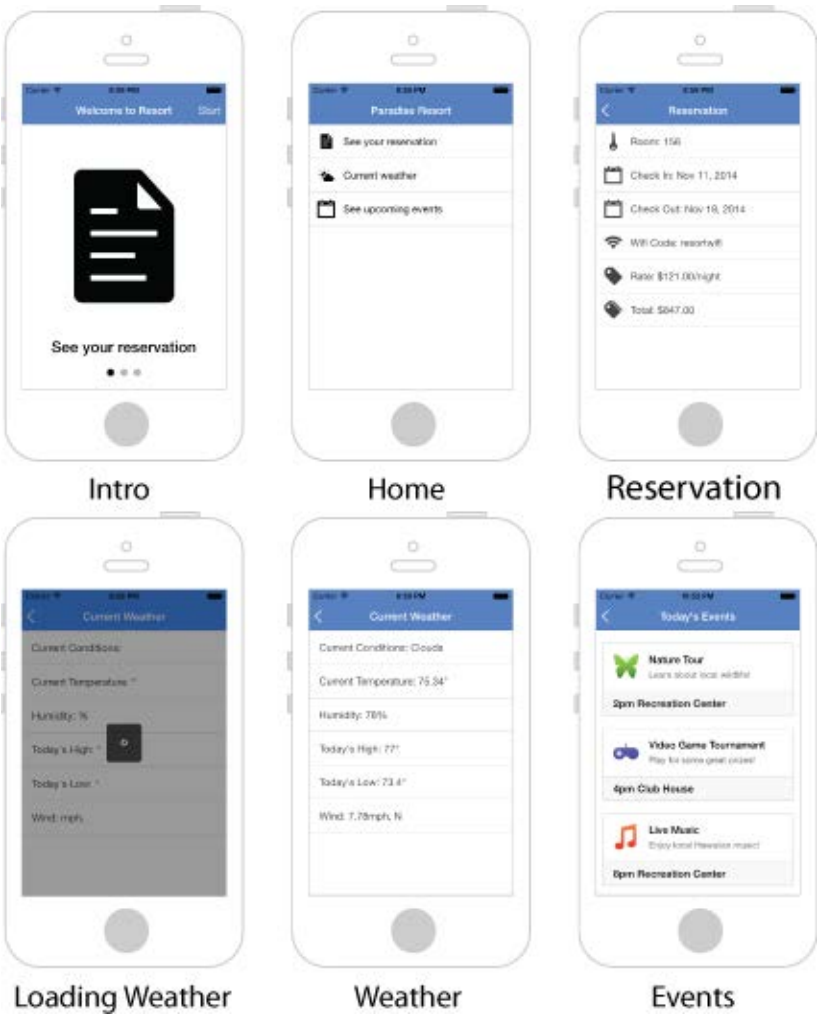


Figure 4.2 – Complete app views and how they will look when the app is completed.

4.2 Setting up the app navigation

Our first task is to get the app navigation setup, and then we'll work on adding the content into each view. Ionic is built to work with an third-party routing framework called ui-router, which is like the central brain for navigation. If you are not familiar with ui-router, I will be showing you the key features we'll need. Ionic layers on top of ui-router in some ways, so you don't often have to worry about the inner workings unless you develop your own custom navigation techniques.

I talk about navigation and routing, which are two different but related concepts. I say navigation as the act of a user moving around inside of an app (a user tapping on a button to go to another place), and routing as the application's internal process to decide what to do when a user navigates (the application deciding where to go when a button has been tapped). In other words, navigation is the user action and routing is the application logic that responds to user input.

Why Ionic uses ui-router and not ngRoute

There is an official router for Angular (ngRoute) which is not used by Ionic. The primary reason is that ngRoute does not support some important features that the ui-router project supports, such as named views, nested views, and parallel views. An example would be allowing a tabbed interface that are actually multiple views. These features built into the core of Ionic, through the `ionNavView` directive. Ionic is built to work only with ui-router, so attempting to use ngRoute will cause issues with your app.

There are a lot of aspects of ui-router that are not covered in this book, and it is best to refer to the website for more detailed information about some interesting features that might be useful in advanced use cases. The website is https://angular-ui.github.io/ui-router/.

In the wireframe in figure 4.1 and in the app overview in figure 4.2, you can see there are really 5 potential places for a user to be. I will refer to these places as views. They are:

- Intro
- Home
- Reservation
- Weather
- Events

Traditional websites have pages, but in a mobile app there are no distinct pages. As a user navigates inside of an app, its less obvious that the view changes, compared to a website where you can watch the URL change in the address bar and see the page reload. I like to think of a view as a well defined visual experience, just as you see in the list above with 5 very distinct places the user can see in the app. Before we setup navigation in our app, I'd like to talk about how to design your app navigation for users.

4.2.1 Designing good app navigation

Mobile app navigation is somewhat like traveling to a new city. Imagine you arrive by train in this new city, and you walk out of the train station near downtown. You may have some idea what you'd like to do or how to get around (such as visit some museums), but you first have to rely on the street signs to understand where you are in relation to your destination. Since you are familiar with street signs and common city rules, you are able to find your way to your destination. In your app, you are responsible for providing the street signs for your users.

Even though we are building a web application inside of our hybrid app, the user experience is not the same as it is for a web application viewed in a browser. Users cannot navigate in the same manner because hybrid apps do not have the browser window and features such as the address bar or back or reload buttons. This means it is up to you, the app developer, to provide the ability to navigate the app.

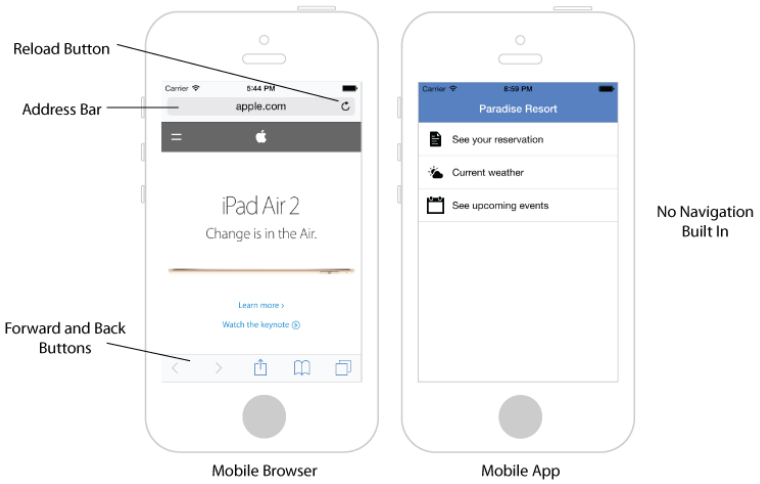


Figure 4.3 – Browsers have navigation aids like address bar, reload, and back buttons. Apps have no buttons and the developer must provide navigation options.

When you consider your navigation, you should consider the flow like in figure 4.1 to understand how the user moves from view to view. There may be multiple ways that a user can get to a given view, but it should always be clear and intuitive for the user. There are many ways to create a navigation flow for users, from custom interactions to the more common features Ionic provides by default. It is best to use common navigation techniques, such as buttons, than to create experiences that a user will have to learn how to use.

I recommend looking at 4-5 of the apps you use on a regular basis and try to understand the navigation flow they use, and what techniques the provide to the user. Do they use buttons? Is there a side menu or tabs to help get to key parts of the app? Do you ever get lost

and if so can you see how? Thinking critically about these items will help you while you design your apps.

4.2.2 Declaring the app views with the state provider

Now its time to dig into some code. Our first task is to add the Ionic navigation components into our app HTML markup. Then we'll declare one view to start with. The result will look like you see in figure 4.4, which is mainly the content container with no content and the navbar.

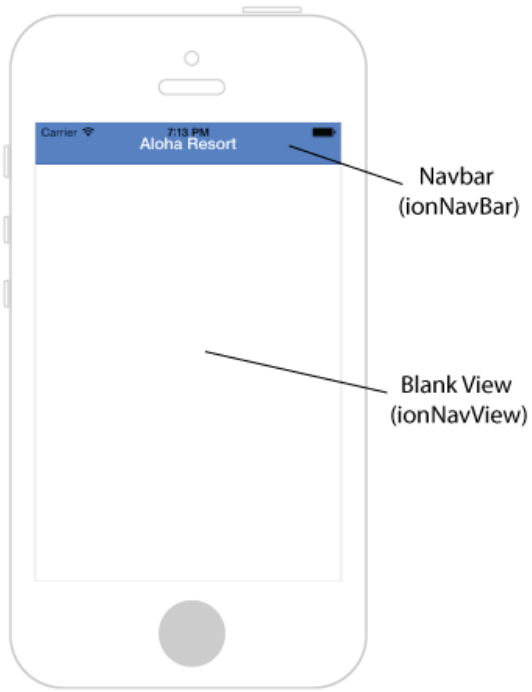


Figure 4.4 –A basic app with navigation and a single view with no content.

The `ionNavView` and `ionNavBar` are the foundational Ionic components for our navigation. `ionNavView` acts like a place holder for different view content to be loaded into the app, where the `ionNavBar` shows a header bar that will update as you move from view to view. These two components are designed to work together, but you could use the `ionNavView` without using the `ionNavBar` if you did not want a top navbar.

If you are familiar with frames in HTML, the `ionNavView` is similar in the sense that it allows content to be loaded inside. However `ionNavView` is not an actual frame. If you want to understand more about Angular and templates, refer back to chapter 3. Remember that Angular has a way to take markup (which we call the template) and inject it into the view (in

this case inside of `ionNavView`). Without `ionNavView`, the app would not know where to load the content, so you will always need at least one `ionNavView` in your app if you use navigation.

The `ionNavBar` will place a top bar in the app, which is found in many apps. It is like your title bar. Think of it like a place that you can put the current view's title and also use it to place buttons, such as a back button. We'll use the `ionNavBackButton` component in this app because our app users will need a way to go back. Looking at figure 4.1 the bottom row views will all have a back button to return the home view. The home view will not show a back button, because it is the top level view and the back button will not display.

In listing 4.1, let's take the index.html file and add in our navigation components. These components cannot run without some JavaScript that we'll add in listing 4.2, but first focus on the components in the markup. This code shows only the contents inside of the body tag of the page, but you will need to retain the rest of the markup in the HTML file.

Listing 4.1 – App navigation components in markup (www/index.html)

```
<body ng-app="App"> #1
  <ion-nav-bar class="bar-positive"> #2
    <ion-nav-back-button class="button-clear"> #3
      <i class="icon ion-ios7-arrow-left"></i> #4
    </ion-nav-back-button>
  </ion-nav-bar>
  <ion-nav-view></ion-nav-view> #5
</body>
```

#1 This is where the angular app is attached to the page, refer to chapter 3
#2 Declare the ionNavBar, here we also give it a style with the bar-positive class
#3 Declare the ionNavBackButton, which will show and hide if there is a way to go back
#4 Use an arrow back icon to be the back button
#5 Declare the ionNavView, this is where each view content will display

So far, we're just declaring place holders for content. We have not actually declared any views, but when we do that these components will automatically know what to do with it. It is likely that many apps you build will have markup similar to listing 4.1 as the foundation for your app navigation.

You can see there are classes on some of the components, and this is common. Ionic allows you to customize the display of many components by using CSS classes. We'll cover these options in more detail later, but I wanted to make sure you knew why those existed.

If you happen to run this code, you'll notice it doesn't really do anything yet. That is because we haven't declared any views. What we actually need to declare is a list of states for our application. States are a concept given to us from the ui-router. A state is a representation of a view, which would contain details such as the url associated with the view, the name of the controller for the view, and the template attached to the view. In this book, we will declare states that are typically linked to a view (the home view in figure 4.1 for example is a state). For a more in-depth discussion about state, you can refer to the ui-router details at https://github.com/angular-ui/ui-router/wiki.

If you'll recall, we talked about how routing is the concept for informing the app what navigation options exist. You could think of it like a folder tree, where states can be organized as children of other states.

Right now we've created the home state but haven't yet declared any of the other files inside of that directory to view. The states we declare include a way to route the user to a particular place in the app, which can be done using URLs or using the name of a state. In this book I'll usually use the state name for navigation.

Let's add some states to our app in listing 4.2. We're going to add these states into the app.js file, which contains our Angular app definition. We will use the `$stateProvider` service to declare our states, and the `$urlRouterProvider` to help provide a fallback in case an invalid request is made. The code in listing 4.2 will be added right after the first line (which is shown).

Listing 4.2 - Declaring app states (www/js/app.js)

```
angular.module('App', ['ionic']) #1
.config(function ($stateProvider, $urlRouterProvider) { #2
  $stateProvider.state('home', { #3
      url: '/home', #4
      templateUrl: 'views/home/home.html' #5
    });
    $urlRouterProvider.otherwise('/home'); #6
})
.run(function($ionicPlatform) { #7
```

#1 Angular module definition, already in the file
#2 Add new config method, and inject $stateProvider
#3 Declare the first state for the home view
#4 Give the state a url that can be used with anchor links
#5 Tell the state to load a template from a given URL when view is active
#6 Declare a fallback url to go to if the app cannot find the requested state
#7 Angular run method, already in the file

Alright we've got our first state declared, and it is called home. It is very simple, it will only try to load a template from the url. As we add more states to the app in this chapter, you will see other configuration values that can be provided. You can always review the full set of options based on the ui-router documentation. The examples of this chapter will demonstrate the most common variations.

The `otherwise` is important because it is able to catch situations where the application is unable to find the requested route, much like a 404 error page on a website. If the user tries to request a state that doesn't exist, the otherwise route will be used to display the home view. It is always a good idea to declare an otherwise in case your app has a routing problem so it can always have something to show instead of a blank page or error. You might consider making a special error view that people can use to send you feedback.

You might have noticed we have a template declared, but we haven't yet created this file. Let's add this last file to make our first view work correctly and see how the view gets loaded into the app. I prefer to organize all of my views into a folder called views, so create a new file at `www/views/home/home.html` and put the contents from listing 4.3 inside.

Listing 4.3 – Add template for home view (www/views/home/home.html)

```
<ion-view title="Paradise Resort" hide-back-button="true"> #1
</ion-view>
```

#1 Use ionView to declare a view template, title is used in navbar and hide-back-button will disable the back button

Now we can run the code and see a it runs correctly. You should see the app running with a blue navbar and the title Paradise Resort. The rest of the view is blank, for the moment. We'll add that in next. It should look like you see in figure 4.3.

Note the `hide-back-button` attribute. The attribute tells the `ionNavBar` that this view does not want the back button to show. There are other `ionView` attributes that you may use that you can find in the documentation.

Now this isn't terribly impressive just yet, we want more! Let's move on to the next part where we'll get the home view setup. Along the way we'll talk about the content area, how to use icons, and lists.

4.3 Building our first view, the home view

The example so far has a blank view with a title, now we'd like to add more content into this view. The primary feature of this page is to provide a list of links that will take us to the weather, events, and reservation views.

In listing 4.3, we have a very basic and very blank view. Inside of this view, we'll put a few things. We'll add `ionContent`, which is a generic wrapper for content but has a lot of features you might not notice immediately. Then we'll create a list of navigational links for each of the views. Lastly, we'll add some icons to make them a little nicer on the eyes. You can preview the result in figure 4.5.

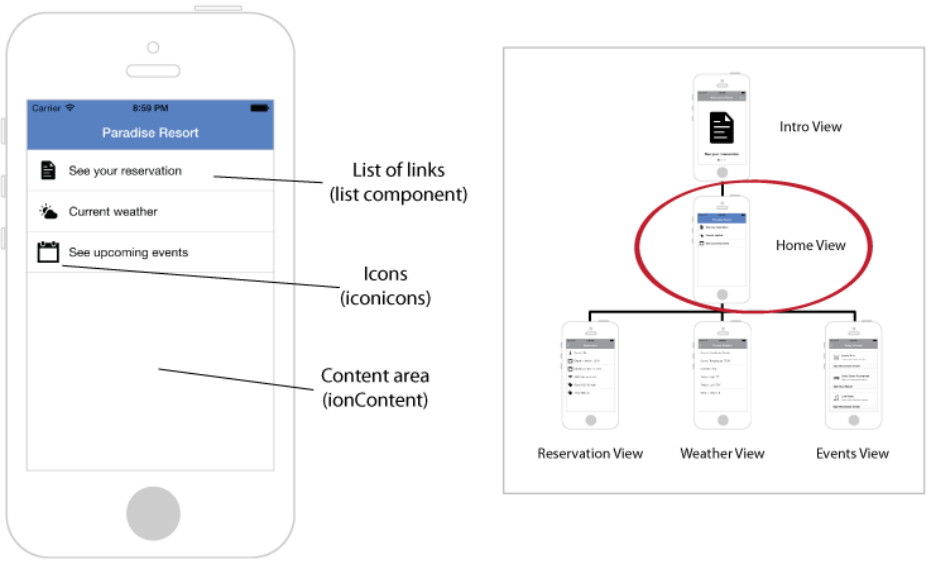


Figure 4.5 – The home view with icons and a list of links and the content is correctly placed below the navbar.

4.3.1 Creating a content container

`ionContent` is the most commonly used content container. It provides a number of features:

- **Sizes content area to device.** It will determine the appropriate height for your content container based on the device.
- **Aware of header and footer bars.** It knows if there are header or footer bars, and will adjust its size and position accordingly.
- **Manages scrolling.** It has a number of configuration options to manage scrolling. For example, you might want to lock scrolling to be one direction only (horizontally), or no scrolling at all.

There are a lot of options with `ionContent`, but they are primarily related to managing the scrolling experience. In most cases you will not need any of those options, but you can see the various options in the documentation. Let's add this tag to our home view. Open the home view file again and add the code like you see in listing 4.4.

Listing 4.4 – Adding ionContent to home view (www/views/home/home.html)

```
<ion-view title="Paradise Resort" hide-back-button="true"> #1
  <ion-content> #2
  </ion-content>
</ion-view>
```

#1 ionView we declared earlier
#2 ionContent that will hold content for the view

That's it? Yes! In this case `ionContent` is pretty easy to use and since we plan to use the default features we don't have to make it any more complex than this. The content area will now resize and take the navbar into consideration when it calculates the size and position of the content. Without it, the content would start in the top left corner behind the navbar, which is obviously not desirable.

Is your content in the wrong place?

The vast majority of the time you will use `ionContent` to wrap your content. If you ever run into trouble where your content is misplaced on the screen, start by double checking that you have `ionContent` in the right place.

There are some situations where you don't want to use ionContent, which you will see one example in chapter 4 where tabs should not be inside of ionContent. Sometimes you may have to add some CSS to make things display like you want if you don't use ionContent. For example, if you use ionHeaderBar without ionContent, the content will be positioned under the ionHeaderBar. Ionic tries to make the design and components work in most cases, but some non-standard use cases require extra CSS.

4.3.2 Using CSS components and adding a simple list of links

Now that we have a content container, we want to add our list of links. Ionic comes with a lot of components that are simply CSS classes applied to elements. If you are familiar with frontend interface frameworks like Bootstrap or Foundation, you will be quite familiar with the method of adding classes to create visual components. Some of the components are mobile focused designs for several form elements like checkboxes, a range slider, buttons, and more.

Ionic has a list component, which is a pair of classes for a list and each list item. The list component has a number of style configurations, and we're going to start with the most basic and then add icons.

Let's add a basic list of links to our app. While you can use a normal unordered list element, I'm actually going to use a div to wrap a list of anchor tags. This is important to note because the CSS styling applied is very complete and will allow you to use the class on different elements.

Listing 4.5 – Adding a list to the home view (www/views/home/home.html)

```
<ion-view title="Paradise Resort" hide-back-button="true"> #1
  <ion-content> #1
    <div class="list"> #2
      <a href="#/reservation" class="item">
        See your reservation
      </a>
      <a href="#/weather" class="item"> #3
        Current weather #3
      </a> #3
      <a href="#/events" class="item">
        See upcoming events
```

```
      </a>
    </div>
  </ion-content>
</ion-view>
```

#1 The ionView and ionContent declared from earlier
#2 Add the list class to a container element to designate a list container
#3 Add the item class to an element to create a list item, in this case a link to another view

Alright, this is getting close. Using a CSS component is normally just following the component guideline and giving elements the proper CSS classes. We'll look at more complex lists in the next chapter, but for displaying a simple list of items this suits our needs quite well. The documentation also shows a number of different list item display types, such as having dividers, thumbnails, or icons.

The list has some links to different URLs, which we have not yet defined. We'll add each view individually, and then the app will be able to navigate to that view. With the item class on the anchor tag, it adopts the list item display. These three links are related to the 3 views, which you can refer back to figure 4.1 to see the app flow.

CSS vs JavaScript components

If you look at the Ionic documentation, the components are split into two distinct categories, CSS and JavaScript. If you look closely, you will notice there are some which appear on both lists, such as header bars and lists. You might wonder why are there two, and is one better than the other?

Some components are CSS only (buttons), others are JavaScript only (infinite scroll), while some are both (tabs). CSS components provide a visual display to a component, but have no real configuration or interactivity. JavaScript components provide more intelligent and interactive components, which may or may not build from a CSS component.

If you ever need to choose between a CSS and JavaScript version of a component, I would recommend using the CSS version unless there is a feature you need in the JavaScript version. While Ionic is very fast, anytime you can reduce the use of JavaScript can help keep the overhead low.

It isn't always obvious, but when a component is both CSS and JavaScript, you can use the CSS classes to modify the JavaScript version. For example in this chapter, the `ionNavBar` is using the CSS class to adjust the color.

4.3.3 Adding icons to the list items

The last thing we'd like to do for this view is add some icons. Ionic comes with a set of icons, called Ionicons, and are bundled by default. Icons are used in many places, so you'll see them frequently. You can view the available icons at http://ionicons.com. The icons are actually a font icon, which are custom fonts that replace standard characters with icons and uses CSS classes to display the icons. If you'd like to use another font icon library (such as Font Awesome) you should be able to include that without conflicts.

The list component has a special display mode that uses icons. Using an extra CSS class and adding an icon element will generate the desired effect of having the icon displayed to the left of the text in the list item. We'd like the icon to be on the left of the text. Let's finish our home view as you see in listing 4.6 by adding some icons and updating the list item with a new class.

Listing 4.6 – Add icons to home view (www/views/home/home.html)

```
<ion-view title="Paradise Resort" hide-back-button="true">
  <ion-content>
    <div class="list">
      <a href="#/reservation" class="item item-icon-left"> #1
        <i class="icon ion-document-text"></i> See your reservation #2
      </a>
      <a href="#/weather" class="item item-icon-left">
        <i class="icon ion-ios7-partlysunny"></i> Current weather
      </a>
      <a href="#/events" class="item item-icon-left">
        <i class="icon ion-calendar"></i> See upcoming events
      </a>
    </div>
  </ion-content>
</ion-view>
```

#1 Add the item-icon--left class to get the desired styling
#2 Add an italics element, but using the icon class with an icon designation to transform it to an icon

This is the most common way to include an icon, but since it is inside of a list we need to use the special item-icon-left class to get the display we desire. You could also use item-icon-right to have the icons float to the right side.

Icons are often declared like this `<i class="icon ion-calendar"></i>`. By default the italics element is an inline element that would modify the text inside. However we have no text inside, just two CSS classes. The first class, icon, gives the element the base CSS styles for an icon, and the second class, ion-calendar, provides the specific icon to display. Together, the inline element becomes an icon. You can see the icon and the entire home view in figure 4.5.

Now the home view is what we want, so let's move on to the reservation view and learn how to display information using a controller.

4.4 Using a controller and model for the reservation view

Very often you will need to add business logic to your view to handle loading data or interactions. I use the term business logic to describe any JavaScript code written that is unique to an app. The home view has no business logic, because it just shows a static list of links. However for our reservation view we will want to be able to load a user's specific data and display it. Since this is just an example, we don't have a real hotel database to use for loading data but we'll still use a controller to contain our data. If you are new to Angular, it is best to review the section in chapter 3 about controllers before continuing with this chapter.

To declare a controller follows the same pattern for any Angular controller. Remember, Ionic is built on top of Angular, so instead of having to provide its own framework it uses

Angular. We will create this new controller, and inside it will contain the model for our reservation. The result of this section is shown in figure 4.6.

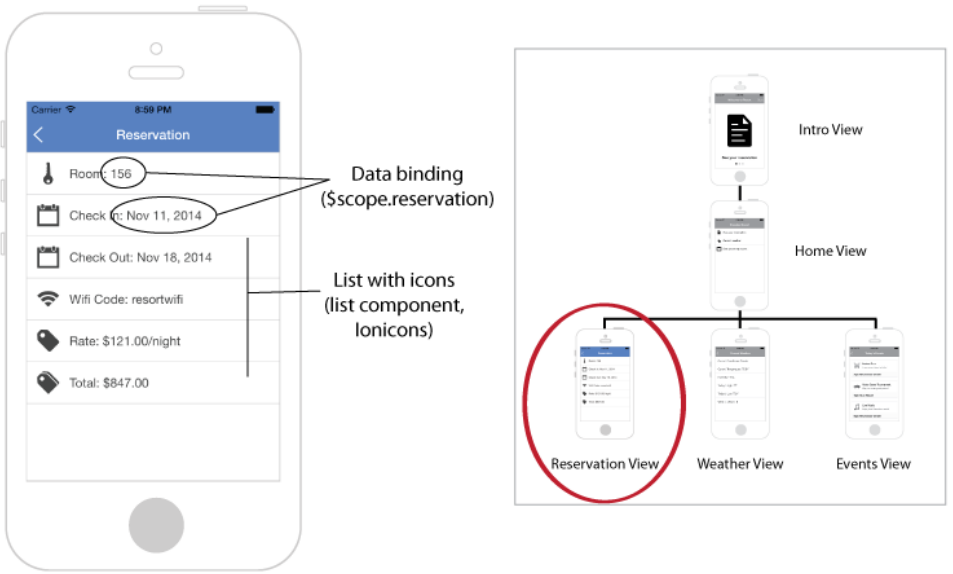


Figure 4.6 – The reservation view, using bindings and loading data from controller.

You'll recall from chapter 3 in order to create a model in Angular, you attach a value onto the $scope object. We'll attach an object with the properties describing the user's reservation details, such as the dates of arrival and departure, room number, and so forth. Let's take a look at our controller. We'll add it to our views directory under `www/views/reservation/reservation.js`.

Listing 4.7 – Reservation controller (www/views/reservation/reservation.js)

```
angular.module('App') #1
.controller('ReservationCtrl', function ($scope) { #2
  $scope.reservation = { #3
    checkin: new Date(), #4
    checkout: new Date(Date.now() + 1000 * 60 * 60 * 24 * 7), #4
    room: 156, #5
    rate: 121, #5
    wifi: 'resortwifi' #5
  };
});
```

#1 Reference our App module
#2 Declare a controller, providing the name and then a function. It takes a list of items to inject, like $scope
#3 Attaching a model object called reservation to the $scope
#4 Setting the dates for the stay, automatically calculating today to next week

#5 Setting other static values for the reservation

This controller doesn't do a whole lot, but it does provide a place for us to hold our data and use Angular's binding features. We didn't do this on the home view, because the list of items on the home view is very simple and unchanging. In this example, you can see how this information would change from user to user, and would be loaded here in the controller.

This controller needs a template like our home.html file above to display information. The reservation view will have a list, similar to the home view, and include some icons into the template.

The major difference in this template is that we'll bind data from the controller into the template. The bindings will also use Angular filters, which is a very useful way to convert data from the model to a different display format. We'll also look a little bit at Angular expressions in action.

Listing 4.8 has our reservation view for the controller we build above.

Note about file organization

Now you may start to see better the reason I prefer to organize my files together by views, it helps to keep related parts together. Many Angular tutorials lump the JavaScript files into one place and the HTML templates in another, making it hard to keep track of related items. Later we will also place CSS files in the folder for a view. Keeping the all the files related to a single view in one location has improved my workflow greatly. I no longer spend precious time trying to find where the related code exists.

You are not required to follow my conventions, but they are my preferred conventions based on my years of building applications with Angular and the conventions I will follow in this book.

Listing 4.8 – Reservation view template (www/views/reservation/reservation.html)

```
<ion-view title="Reservation"> #1
  <ion-content> #2
    <div class="list"> #3
      <div class="item item-icon-left"> #4
        <i class="icon ion-key"></i> Room: {{reservation.room}} #4
      </div> #4
      <div class="item item-icon-left">
        <i class="icon ion-calendar"></i> Check In: {{reservation.checkin |
      date:'mediumDate'}}
      </div>
      <div class="item item-icon-left"> #5
        <i class="icon ion-calendar"></i> Check Out: {{reservation.checkout |
      date:'mediumDate'}} #5
      </div> #5
      <div class="item item-icon-left">
        <i class="icon ion-wifi"></i> Wifi Code: {{reservation.wifi}}
      </div>
      <div class="item item-icon-left">
        <i class="icon ion-pricetag"></i> Rate: {{reservation.rate | currency}}/night
      </div>
```

```
      <div class="item item-icon-left"> #6
        <i class="icon ion-pricetags"></i> Total: {{reservation.rate * 7 | currency}}
       #6
      </div> #6
    </div>
  </ion-content>
</ion-view>
```

#1 Declare our view with the title Reservation
#2 Add a content wrapper to help with content display
#3 Wrap the list with the list component class
#4 A list item with an icon, and binds the room value into the template
#5 A list item using a filter in the binding, in this case formatting the date
#6 A list item using a binding with an expression and filter

At first glance this may appear very similar to the home view, but we've used Angular bindings to add data from our model in the controller (`$scope.reservation`) and display it in the template. This is very common in Angular, and you will likely use this frequently. Anything between the `{{}}` is evaluated as a special type of concept called an Angular expression. Angular expressions allow you to bind data from the $scope and even write math expressions that will be evaluated automatically. Refer to chapter 3 for a deeper explanation of Angular expressions and data binding.

Let's take a closer look at the example `{{reservation.rate * 7 | currency}}`. First there are two key parts, separated by the pipe (|). On the left we have the expression, and on the right we have a filter. In the expression, we are able to do math by multiplying the daily rate by 7 to get the weekly rate. When an expression has a variable name, such as `reservation.rate`, it tries to resolve the value by looking at the $scope for that property. If it doesn't exist, the expression will fail and display nothing. Now the filter is optional, but here we use the built in Angular currency filter, which takes a value and formats it for the local currency of the browser. It does not change the value of the `reservation.rate` property to include the currency sign, it just transforms it for the display.

While there are many other tricks and possibilities with expressions, the most common use is simply binding data to the view. We'll see them again in action with the weather view. Before we move to another view, we haven't yet added this view to our application's state provider. At this point we have the files in place, but haven't told the application about it. Open up the app.js file again where we declared our first state, and let's add a new state. This will be placed right after the existing home view, and be sure that there is no semicolon between them.

Listing 4.9 – Declaring the reservation state (www/js/app.js)

```
.state('reservation', { #1
  url: '/reservation', #2
  controller: 'ReservationCtrl', #3
  templateUrl: 'views/reservation/reservation.html' #4
});
```

#1 Declare a new state called reservation
#2 Route the app using the /reservation url
#3 Declare the name of the controller used for this view

#4 Declare the view file to load

Great, now we have our view declared and everything should work, right? Not just yet, one easy to forget step is left. We are still building web applications, so we created a new JavaScript file but haven't yet added it to the index.html file to load. If you see errors in the JavaScript console that says the `ReservationCtrl` is undefined, then you haven't included that file in your app or there is a syntax error somewhere. Add this line to your index.html, right before the `</head>`.

```
<script src="views/reservation/reservation.js"></script>
```

Now we can run our app and tap on the reservation link to view the reservation details. It should appear like in figure 4.5. You'll notice the back button appears, and as you navigate to child views the back button will automatically show and hide. We've hidden it from the home view, so the child views like reservation will show the back button. Just a note, if you refresh the app while still on the reservation view the back button will not appear. This is because it is like your first visit to the app, and there is no place to go back to in the history. If you get stuck, you can change the URL in the browser address bar to http://localhost:8100 to start over.

4.5 Loading data into the view, the weather view

When you are at a tropical resort, it is good to know that your weather forecast is sunny and warm. You came for the beach, so it better be beach weather! In this next view, we will load weather data from an external service. There are many services that provide this data, but in this case I am using Open Weather Map, which has a free and open API. Other services exist but may require an account or payment to load data.

Like the reservation view, we will need an Angular controller. This controller will take care of loading the data from Open Weather Map, and then store it on our model so the view can bind data. I will use the `$http` service from Angular to handle the data loading.

The view will be another list of items, such as current temperature and conditions, today's highs and lows, and wind speed and direction. I'll show you how you can use $scope methods in your template expressions to calculate information. In this case I will take the wind direction that is given to us in number of degrees and convert that into compass value such as North, East, South, West.

Since we are loading data from an external website, this can take some time to happen. So far a view is displayed immediately upon navigation to that view, but in this case we cannot show the weather data until it has loaded. This may take under a second, or in some cases can take a few seconds, depending on the speed of the connection, the server response speed, and other variables (many out of your control). In order to provide a better experience, I will show you how to use the `$ionicLoading` service to display a loading indicator while the data loads.

Figure 4.7 shows the result of the loading component in action and then display of the local weather details.

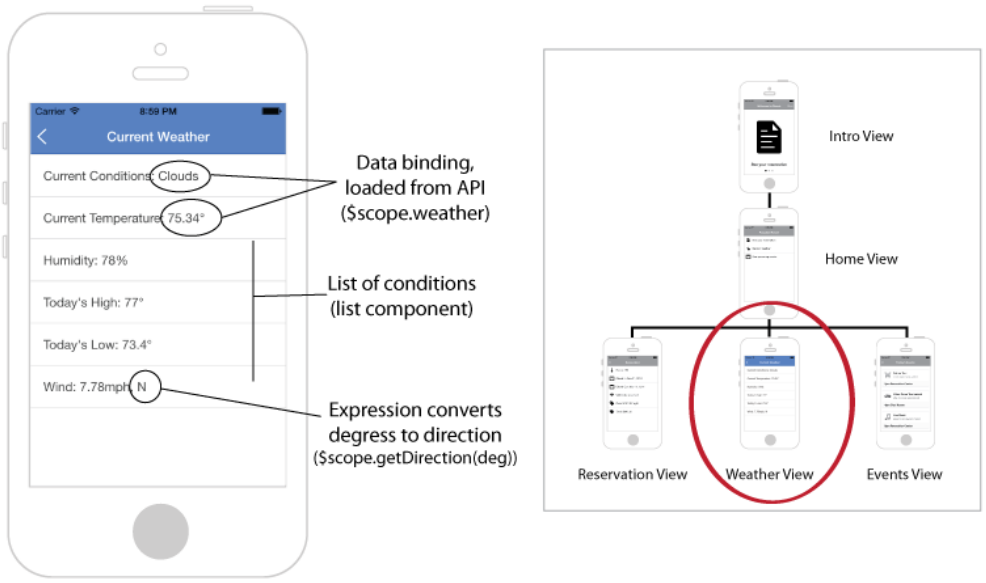


Figure 4.7 – Weather view, demonstrating the loading on the left and after load on right

We'll start by adding the template file, then add the controller and data loading steps, and lastly implement the loading component.

4.5.1 Adding the template for the weather view

The template for the weather view is fairly simple, we just want to show a list of weather conditions. We'll create a list, and bind the values for the data into it. This will be mostly similar except for one new type of Angular expression that I'll cover.

Create a new file for the weather view, using the same conventions at `www/views/weather/weather.html`, and add the code from listing 4.10.

Listing 4.10 – Weather view template (www/views/weather/weather.html)

```
<ion-view title="Current Weather"> #1
  <ion-content> #2
    <div class="list">
      <div class="item">Current Conditions: {{weather.weather[0].main}}</div> #3
      <div class="item">Current Temperature: {{weather.main.temp}}&deg;</div> #3
      <div class="item">Humidity: {{weather.main.humidity}}%</div> #3
      <div class="item">Today's High: {{weather.main.temp_max}}&deg;</div> #3
      <div class="item">Today's Low: {{weather.main.temp_min}}&deg;</div> #3
      <div class="item">Wind: {{weather.wind.speed}}mph,
       {{getDirection(weather.wind.deg)}}</div> #4
    </div>
  </ion-content>
</ion-view>
```

#1 Declare our view and title for the navbar
#2 Wrap the content in a container
#3 Add list items that bind to data properties of the weather object
#4 This item has two bindings, the second calls a method on the scope

Here we create a new view and give it a title for Current Weather, and use the content container to manage the position of our content area. I've also used the list CSS classes again, just to keep it simple, as a way to list the weather details.

This template has more complex bindings because the data returned by the Open Weather Map is formatted as a JSON string which is parsed into a JavaScript object by Angular. You can view the standard output of the weather data by viewing the API in your browser http://api.openweathermap.org/data/2.5/weather?q=London,uk. You can replace the London,uk value with any city to load data matching that query. If you review this output from the API, you can see the object and array items that we have to navigate to access specific data values.

I wanted to point out one more expression that is a little different from the rest. If you look at the last list item you will see `{{getDirection(weather.wind.deg)}}`. This expression is actually going to reference a method attached to the $scope in our controller. We have not written this yet, but the method is called `getDirection` and takes a single parameter, which is the wind direction in degrees. You can use methods like this in an expression as part of your logic when necessary.

4.5.2 Create weather controller to load external data

Now let's setup the controller and load some data. We will use the `$http` service from angular to load data from a url. We'll inject the `$http` service into the controller, get a url, and then handle the success or failure of the HTTP request. Open a new file for the controller at `www/views/weather/weather.js` and put the contents of listing 4.11 inside.

Listing 4.11 – Weather view controller (www/views/weather/weather.js)

```
angular.module('App') #1
.controller('WeatherCtrl', function ($scope, $http) { #2
  var directions = ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW']; #3

  $http.get('http://api.openweathermap.org/data/2.5/weather?lat=21.873457&lon=-
      159.453314&units=imperial') #4
.success(function (weather) { #5
    $scope.weather = weather; #6
}).error(function (err) { #7
     #7
  }); #7

  $scope.getDirection = function (degree) { #8
    if (degree > 338) { #9
      degree = 360 - degree; #9
    } #9
    var index = Math.floor((degree + 22) / 45); #9
    return directions[index]; #9
  };
});
```

#1 Reference the angular module for our app
#2 Declare the controller and inject the $scope and $http
#3 An array with the possible directions for wind
#4 Make an HTTP request to load the data at the url given
#5 Handle a successful response, and get the weather object returned
#6 Assign weather data to the $scope.weather model
#7 Error handling will be done here later
#8 Method used to convert a value of degrees to a cardinal direction from directions array
#9 Calculate which of the directions the wind is blowing.

This controller will now automatically load the weather data every time the user loads the view. It is loaded into the `$scope.weather` model, so our template can then bind to that data. You can use the browser developer tools to inspect the response from the API to see what has been sent back. This is the most basic way to load data into your application from a URL.

We are not handling the error yet in case the API doesn't send back data. This will be added in the next section, but you should always have an error handler with `$http`.

The other major section is the `getDirection` method, which takes a number and returns one of the values from the direction array. This is done because we want to convert data from the model when it is displayed. This would be more appropriately done as an Angular filter, but for the purposes of this example I used this approach.

We need to add this new view to our list of states, and also include the script tag for the new controller. Open the main app file and let's add another state to the end of our state list.

Listing 4.12 – Declare the weather view state (www/js/app.js)

```
.state('weather', { #1
  url: '/weather', #2
  controller: 'WeatherCtrl', #2
  templateUrl: 'views/weather/weather.html' #2
});
```

#1 Declare the weather state, add it to the existing
#2 Add the url, controller, and template values to define the state

Then also add the script tag to load the weather controller into the index.html file, right before the closing `</head>` tag.

```
<script src="views/weather/weather.js"></script>
```

Now you can preview using ionic serve to view the app in its current state. Select the weather link, and it will open the weather view. You will notice the view will load and the bindings will be empty for a brief moment until the data has loaded. This is not very pretty, and could confuse the user. Adding a loading indicator is important for the user experience, so let's do that now.

4.5.3 Adding a loading indicator to the weather view

The loading screen prevents the user from using the app until the loading has finished, so it is important to consider when this is appropriate or when it might unnecessarily stop the user

from interacting with the app. If your app cannot continue until data is loaded, then it is likely this component will come in handy. For example, if you are loading account data when the app opens, the loading component can display since the user cannot act on their data yet. You can see the default display in figure 4.8.

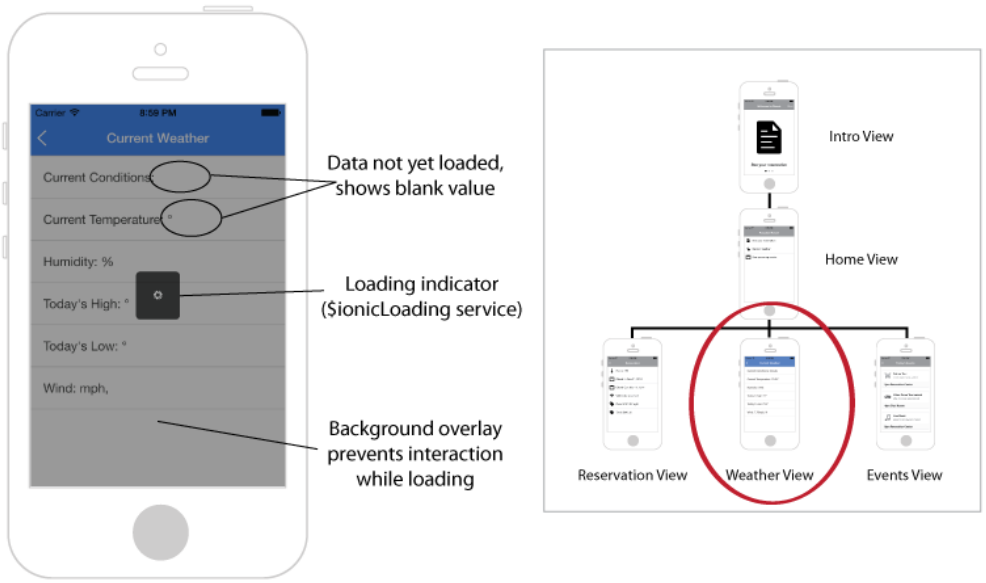


Figure 4.8 – Weather view with loading indicator active while data is loaded from API.

The loading component has two methods, `show()` and `hide()`. You will have to tell the loading component when the hide, because it will not automatically know when loading is finished. You can view all of the configuration options on the documentation.

In our example, we will want to show the loading indicator while the HTTP request is waiting to finish. We will need to tell it to show right before we make the request, and then tell it to hide when the response is returned. The loader will not hide automatically, because it is unable to determine when it is the correct time to hide automatically.

The loading component is implemented in the controller using JavaScript only, so open up the weather controller again and let's update it. I'll only modify the relevant parts.

Listing 4.13 – Adding loading component to weather (www/views/weather/weather.js

```
.controller('WeatherCtrl', function ($scope, $http, $ionicLoading) { #1
  var directions = ['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW'];

  $ionicLoading.show(); #2

  $http.get('http://api.openweathermap.org/data/2.5/weather?lat=21.873457&lon=-
      159.453314&units=imperial').success(function (weather) {
```

```
    $scope.weather = weather;
    $ionicLoading.hide(); #3
  }).error(function (err) {
    $ionicLoading.show({ #4
      template: 'Could not load weather. Please try again later.',#4
      duration: 3000 #4
    }); #4
  });
```

#1 Inject the $ionicLoading service into the controller
#2 Show the loading component, call it right before HTTP request starts
#3 When response is successful, hide the loading component
#4 If there was an error, this uses the loader to display a message and closes after 3 seconds

Take a look at the updated controller now, and you will notice how the loading component is shown and hidden based on when we tell it. We have to first inject the loading service into our controller. Since Ionic's services are built on top of Angular, they are injected just like any other Angular service (like `$http`).

The show is used right before an asynchronous command is executed. HTTP requests made by JavaScript are always asynchronous, and the success or error methods allow us to handle what to do when the HTTP request has finished. Hopefully it will be successful and we just hide the loading component, but in case of an error we reconfigure the loading component to show an error message.

The second show method will automatically close after 3 seconds, since we are using here to display an error. It accepts an object containing configuration values to modify the behavior, which you may wish to use to customize the loading message for example.

This also demonstrates that the loading component acts like a singleton, or in other words there is always only one loading component. The second show method just updates the already visible component with a new configuration. It does not create two loading components. This is important in cases where you have multiple asynchronous events happening, you will have to design logic to properly handle when the hide the component. For example, if you are displaying a chart that has to load data from two separate HTTP requests, you will have to decide if both requests have to finish before you hide the loading component, or if you hide it as soon as the first request finishes so you can chart the data immediately and add the second set of data when it finishes later.

You should also consider using other components to give the user feedback about errors when they occur that we'll cover in the next few chapters. You may want to do different things depending on the type of error, for example if the weather API sends back a message saying it is temporarily down, you could try to reload the data again after waiting a moment.

Next up is the events list view, and we'll spend some time looking at another useful visual component.

4.6 Create cards for the events view

Cards are really just special lists, as you will see in listing 4.14. The card component is used fairly often in many apps because they are good at displaying each item in a clean format with

some nice visual depth. They are best used for displaying a piece of data with some visual separation from the rest of the content.

The resort has daily events, and in our app we'd like to be able to show the daily events. In this example we'll just keep it simple and show a static list, but you could imagine this list being loaded updated daily and loaded like in the weather component. I want to show you the card component, which is based on the list component and displays information with a little more design. You can see the view in figure 4.9.

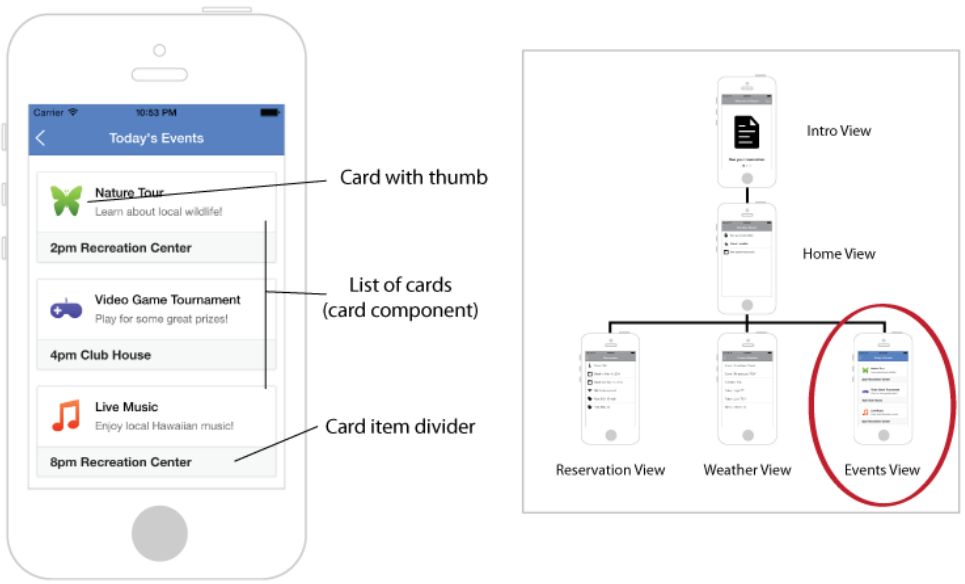


Figure 4.9 – Events view with cards to show upcoming events.

This is a nicer display for our events, and it takes a few CSS classes and elements to get these cards to display. You can review all of the ways you can style cards on the documentation, and here we'll use the avatar display mode. Let's start this view by creating the template in `www/views/events/events.html`.

Listing 4.14 – Events view template (www/views/events/events.html)

```
<ion-view title="Today's Events"> #1
  <ion-content> #1
    <div class="list card"> #2
      <div class="item item-avatar">
        <img src="img/butterfly.png" />
        <h2>Nature Tour</h2>
        <p>Learn about local wildlife!</p>
      </div>
      <div class="item item-divider">2pm Recreation Center</div>
    </div>
```

```
    <div class="list card">
      <div class="item item-avatar"> #3
        <img src="img/video-game.png" /> #3
        <h2>Video Game Tournament</h2> #3
        <p>Play for some great prizes!</p> #3
      </div>
      <div class="item item-divider">4pm Club House</div>
    </div>
    <div class="list card">
      <div class="item item-avatar">
        <img src="img/music.png" />
        <h2>Live Music</h2>
        <p>Enjoy local Hawaiian music!</p>
      </div>
      <div class="item item-divider">8pm Recreation Center</div> #4
    </div>
  </ion-content>
</ion-view>
```

#1 Declare the view and content wrappers
#2 A card class is added with a list class, and contains elements with the item class
#3 The first list item in the card is an avatar type, which formats the image and text automatically
#4 The list divider is a way to emphasize text

There are images loaded here, which you need to download from the GitHub repository here: https://github.com/ionic-in-action/chapter4/tree/master/www/img. Download and save each image into the `www/img` directory of your project.

Before we can see this in action, we need to add our view. Like usual, the app.js file must be updated with a new state for our events view as you see in listing 4.15.

Listing 4.15 – Events view state (www/js/app.js)

```
.state('events', {
  url: '/events',
  templateUrl: 'views/events/events.html'
});
```

Now we can run our app and view the list of events with a visually appealing card design for each event. The last goal is to add an introduction to our app, which will include a set of slides that give a brief introduction and tour of what the app is about. Let's look at how to use a slide box component and build out this tour!

4.7 Using the slide box component for app intro tour

The resort wants to make sure that the first time someone uses the app that the user is able to see a quick tour of what the app can do for them. There are many ways to do this, but I've decided to use the `ionSlideBox` component to show a simple slideshow of the 3 primary features.

Slide boxes are used in many places and many ways. They are great for showing items in a list that you can swipe between, such as a list of images for a product or to have a rotating view of suggested items. The slide box is able to automatically run like a slideshow or allow the user to swipe between the items.

There is `$ionSlideBoxDelegate` service that can be used to programmatically control the slide box. For example, you could have a button that could use the slide box service to force the slide box to go to a particular slide. This service is not used here, but is helpful in cases where you need to add more control to how the slide box operates. It is also possible that you might use the slide box multiples on the same view, and in that case you can control each of them independently. The slide box service is able to name each slide box instance, and then target each individually.

In this example, the tour will show 3 slides using the slide box. I'll apply a little bit of extra CSS to make the display work as intended, because by default a things will only size to the default size of the content and we want the slides to be full screen. You can see the slide box in action in figure 4.10.

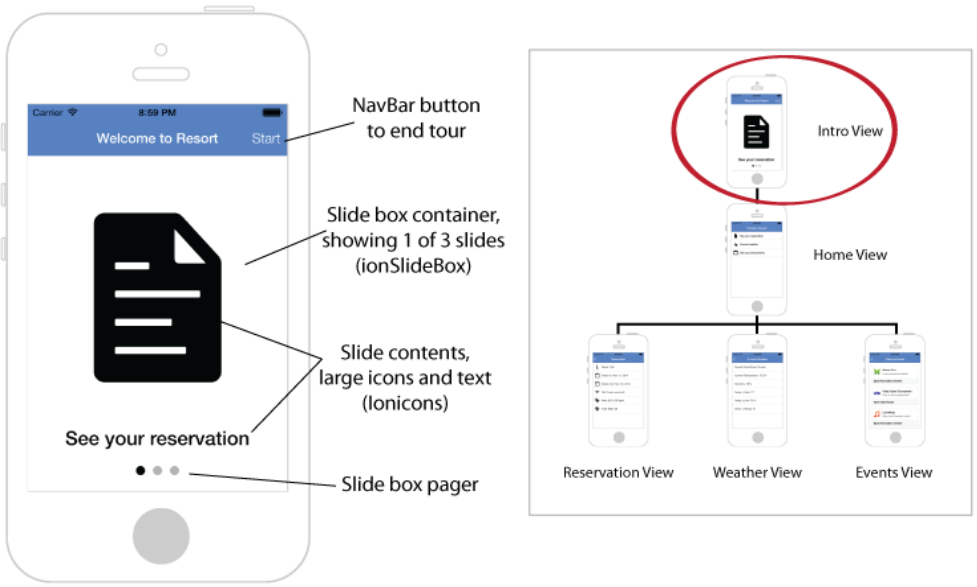


Figure 4.10 – Slide box component used in the tour view.

Let's take a look at how to add a slide box, which in many cases uses just markup to display the component. Listing 4.16 shows the template for our view which you should put into a new file at `www/views/tour/tour.html`.

Listing 4.16 – Tour view template (www/views/tour/tour.html)

```
<ion-view title="Welcome to Paradise Resort" id="tour-view"> #1
  <ion-nav-buttons side="right"> #2
    <a class="button button-clear" href="#/home" nav-clear>Start</a> #2
  </ion-nav-buttons> #2
  <ion-slide-box> #3
```

```
    <ion-slide> #4
      <span class="icon icon-slide ion-document-text"></span> #4
      <h3>See your reservation</h3> #4
    </ion-slide> #4
    <ion-slide>
      <span class="icon icon-slide ion-calendar"></span>
      <h3>Find upcoming events</h3>
    </ion-slide>
    <ion-slide>
      <span class="icon icon-slide ion-ios7-sunny"></span>
      <h3>Get the weather</h3>
    </ion-slide>
  </ion-slide-box>
</ion-view>
```

#1 Declare our view, and give an id so we can target CSS
#2 Add a button to the navbar that allows the user to go to the home view
#3 The ionSlideBox acts as our content wrapper and slide box container
#4 Each ionSlide is automatically added as a new slide to the slide box

The names of the tags for the slide box are very clear, which helps developers understand what is happening. The slide box will have 3 slides, and each contains an icon and a header tag with some information about the app. The slide box will only be as large as the content inside of it is calculated, and right now since our icon and header tags are standard HTML elements, they will only make the slide box as tall as the text itself.

In our case we'll want to enhance this with some CSS styling. In this example I've added an id to the `ionView`, which will be used next when I add some CSS. I find it helpful to prefix CSS by view so I can ensure that my styles will not affect other areas of the app. However you can write your CSS selectors however you prefer.

Let's add that CSS to our app now. Open up the `www/css/style.css` file, which is provided by default and is currently blank. I have three style blocks to add to give our tour the design we want.

Listing 4.17 – CSS for tour view (www/css/style.css)

```
#tour-view .slider { #1
  height: 100%; #1
} #1
#tour-view .slider-slide { #2
  padding-top: 100px; #2
  text-align: center; #2
} #2
#tour-view .icon-slide { #3
  font-size: 20em; #3
  display: inline-block; #3
} #3
```

#1 Styles to make our slider full height
#2 Provides padding for the top of the slide and center the content
#3 Make the icons large and display as inline blocks

The CSS here makes our icons large and centers the content, as well as making the slide box full height. This is helpful so the user can swipe on the whole screen (except the header bar) in order to change the current slide, otherwise they would have to swipe only on the content itself. You could improve or modify this styling to your own needs, and depending on the content that you displayed it might changed dramatically.

Our last step for the app is to update the state with our new tour view, and then we want to change the default state to the tour for when the app is first opened. Open up again the app.js file and we'll add a new state, and change the default otherwise route.

Listing 4.18 – Tour state and update default route (www/js/app.js)

```
.state('tour', { #1
  url: '/tour', #1
  templateUrl: 'views/tour/tour.html' #1
}); #1

$urlRouterProvider.otherwise('/tour'); #2
```

#1 Add the tour state with a route and template
#2 Replace existing otherwise route to be the tour

With that, our app is complete! If you launch the app from scratch, it will now first take you to the tour and then you can go to the home view. If you have been using the live reload feature this entire time, you may not be redirected to the tour. If that is the case, clear the value after the pound symbol in the url so it is just `/#/`. This will reset the route and take you to the tour.

4.8 Chapter challenges

Now that you've covered a lot about how to build a navigable interface for your mobile app, here are a few challenges to improve chapter example into a more comprehensive app.

- **Add a new state.** Try to add a new state to the app. For example, add another state with a view containing directions to the resort.
- **Improve the design.** Get creative and improve the display of the weather view. Look at weather apps for some inspiration.
- **Implement the wind direction filter.** In the weather view, replace the getDirection method in the scope with a filter that can take a degree number and return a string.
- **Cache weather data.** Instead of always requesting new weather data, find a way to cache and only reload the weather after if its more than 15 minutes old. Consider using localStorage to store the data.
- **Create a weather service.** This demo uses $http in the controller to load data. Try to build a weather service for the loading of data so the controller doesn't use $http.

4.9 Summary

This chapter covered the primary means for navigation inside of Ionic apps and a number of the components available. We covered the following topics.

- Ionic apps are built around the idea of states. States are declared in the `config()` method using the `$stateProvider`.
- Ionic has the `ionNavView` tag, which is used to load templates into when states change.
- The `ionNavBar` tag automatically can update the title of the navbar based on the current view. It can also contain `ionNavButtons`, which can add buttons to the right and/or left side of the navbar.
- The list and card components are mobile friendly ways to display lists of content.
- You can load data using the $http service into your controller, and use the `$ionicLoading` service to show a loading indicator while it loads.
- The `ionSlideBox` is a fully featured slideshow component for a mobile interface, and we used it as an app introductory tour.

5

Using Tabs for Navigation

In this chapter you will learn how to:

- Use the tabs component with individual navigation histories
- Display a list of items that can be toggled and reordered
- Use a pull to refresh technique to reload data
- Use several mobile form controls

This chapter is a continued look at many of Ionic's features, and just like in chapter 4 we will be building a complete app from start to finish. We will be building a mobile app to show current market and historical chart data for Bitcoin in many different currencies. The interface will leverage the Ionic tabs component to have 3 tabs: a tab to view current market rates, a tab to view a chart of historical rates, and a tab for currency management.

You will learn more about how to have a navigation window inside each tab. This is important when you want to create richer experiences with tabs that maintain the UI state between tabs. Also when you load data from an external source, the data is cached even if you change between tabs, improving speed and avoiding unnecessary HTTP requests.

What is Bitcoin?

Bitcoin is a popular digital cryptocurrency. It has a buy and sell price, much like a stock or commodity, and is exchanged via digital marketplaces. For our purposes, we are mostly interested in the current price of Bitcoin compared to traditional currencies (such as the USD or EUR) and also interested in visualizing the price history over the past month.

You can read more about Bitcoin and the technology that powers it at https://bitcoin.org.

In figure 5.1 you will see the app that we will create. We will show the current rates for Bitcoin in several currencies, which will compare the current price over the past 24 hours to indicate if it is trending positively or negatively. Then we'll show the historical price, averaged hourly, over the past month. We'll use a third party library to generate the chart. Lastly, the currencies that are displayed in the app can be configured by toggling them to show or hide them, as well as the ability to reorder so your favorite currencies are at the top.

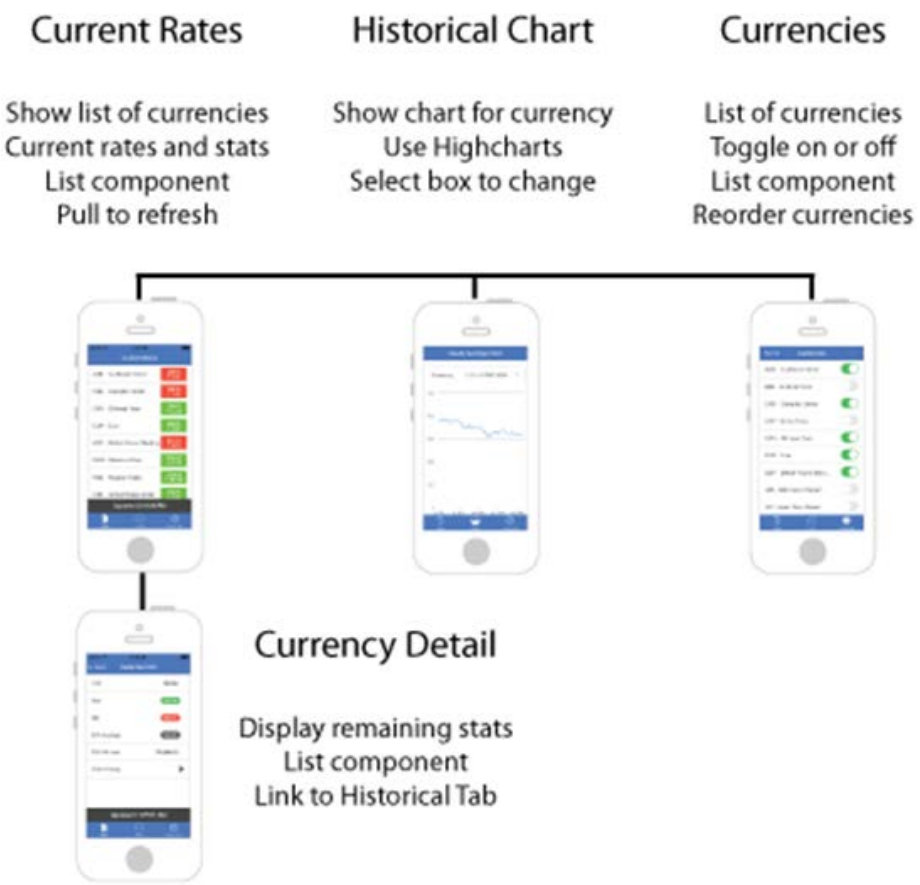


Figure 5.1 – Bitcoin app example, with 3 tabs and 4 views

The entire example is available on GitHub at https://github.com/ionic-in-action/chapter5. The example is also live at https://ionic-in-action-chapter5.herokuapp.com.

5.1 Setup chapter project

You can follow along in this chapter by either creating a new Ionic app and adding the code from the listings in this chapter, or by cloning the finished app from the Ionic in Action GitHub repository and following along with each step. At the end use `ionic serve` to preview the app in a browser.

5.1.1 Create a new app and add code manually

To create a new project for our app using the Ionic CLI utility, open the command line and execute the following command. Remember, you can refer to chapter 2 if you need to refresh how projects are setup.

```
$ ionic start chapter5 https://github.com/ionic-in-action/starter
$ ionic serve
```

5.1.2 Clone the finished app and follow along

To checkout the finished app and use Git to follow along for each step, use the following command to clone the repository and checkout the first step.

```
$ git clone https://github.com/ionic-in-action/chapter5.git
$ cd chapter5
$ git checkout -f step1
$ ionic serve
```

5.2 ionTabs: Adding tabs and navigation

Our first task is to add in our base navigational elements, the ionNavBar and ionNavView components. The ionNavBar will be useful to dynamically update the title bar depending on the tab we are using, and the ionNavView will contain the tabs template. We saw these two working in the chapter 4 app, if you need a refresher on their purpose. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step2
```

In this section we will implement the basic tabs and navigation, as you can see in figure 5.2.

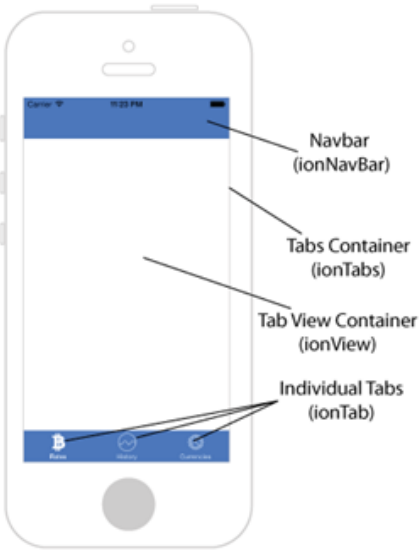


Figure 5.2 – App with tabs, base navigation, and blank content

In listing 5.1, we will update the `www/index.html` file with the navigation components.

Listing 5.1 – Adding ionNavBar and ionNavView (www/index.html)

```
<body ng-app="App"> #1
  <ion-nav-bar class="bar-positive"> #2
    <ion-nav-back-button class="button-clear"> #3
      <i class="ion-chevron-left"></i> Back #3
    </ion-nav-back-button> #3
  </ion-nav-bar>
  <ion-nav-view></ion-nav-view> #4
</body>
```

#1 The body element has the ngApp attatched to it
#2 Add the ionNavBar component and give it a style
#3 Add ionNavBackButton to show or hide during navigation
#4 Add the ionNavView component

This places the components into our templates so they will be able to render our routes. The back button component is in place for later when we have a view that can navigate into child views. Now we have to declare a route and a template before anything will appear.

Open up the www/js/app.js file so we can declare our first route. Modify the existing contents to add the config method as you see in listing 5.2.

Listing 5.2 – Add first route to app config (www/js/app.js)

```
angular.module('App', ['ionic']) #1
.config(function ($stateProvider, $urlRouterProvider) { #2
```

```
  $stateProvider
    .state('tabs', { #3
      url: '/tabs', #3
      templateUrl: 'views/tabs/tabs.html' #3
    }); #3
  $urlRouterProvider.otherwise('/tabs'); #4
})
```

#1 Declare the App module, and include the ionic module
#2 Declare the config method and inject services
#3 Declare a single state for the tabs
#4 Set a default route

Now we have our route declared and a default route set when no other matches are found. Before we can preview our app, we need to add the tabs template.

5.2.1 Adding tabs container and three tabs to the app

Tabs are very common in mobile apps, and Ionic provides a feature rich component for you to create them quickly. Tabs are commonly used to show a visual connection between several views. There is no actual limit on the number of tabs you could use, but typically 2-5 tabs are used due to the limited space available. Tabs can be used nearly anywhere in your app, except inside of an ionContent directive due to a CSS collision that can occur when ionTabs is placed inside ionContent.

Ionic provides two components for building your own tabs, ionTabs and ionTab. Much like ionSlideBox, you declare first the ionTabs and inside you can have as many ionTab components as you need. In our case, we will declare three tabs.

The tabs can have an icon, a title, or both. You can modify the way the titles and icons appear by applying different classes, and in our case we will apply a class to have the title show with an icon above. Tabs also can have an active and inactive icon state, which we will use different icons depending on if the tab is active or not.

Let's add our template with the tabs to the app. Create a new file at www/views/tabs/tabs.html and include the content from listing 5.3 in it.

Listing 5.3 – Tabs template (www/views/tabs/tabs.html)

```
<ion-tabs class="tabs-icon-top tabs-positive"> #1
  <ion-tab title="Rates" icon-on="ion-social-bitcoin" icon-off="ion-social-bitcoin-
      outline"> #2
  </ion-tab> #2
  <ion-tab title="History" icon-on="ion-ios7-analytics" icon-off="ion-ios7-analytics-
      outline"> #2
  </ion-tab> #2
  <ion-tab title="Currencies" icon-on="ion-ios7-cog" icon-off="ion-ios7-cog-outline">
      #2
  </ion-tab> #2
</ion-tabs>
```

#1 Declare the ionTabs component to wrap all tabs and give it a class to modify title and icon display
#2 Declare the tabs with icons for active and inactive and titles

Tabs are really pretty simple to declare, with the title the only attribute you must declare for each tab. Right now they are empty, but if you preview the app in the browser now you will see the tabs along the bottom like in figure 5.2. You can click on each tab and see the icon state change to indicate the active tab.

Now before we start to add content into our tabs, we will want to setup each tab with its own ionNavView.

5.3 Adding ionNavViews for each tab

Right now our tabs are empty, and we want to use additional `ionNavView` components to load our components. This is important because it will allow each of the tabs to maintain its own navigational history. It allows you to use a back button that is only for a given tab instead of the whole app. In figure 5.3 you can see how the user experience would flow with each tab having its own navigational history. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step3
```

Figure 5.3 – Flow of a user through tabs with individual navigational histories

Tabs don't require individual views

The technique of using ionNavView element inside of each tab is not always necessary. I believe it provides a cleaner structure and separation for our app, as well as the ability to have individual view histories.

Imagine you have a single view with each tab containing a list of settings. In that case, the tabs would not have navigational abilities and the tabs are used primarily as a way to separate the settings into meaningful groupings.

My general recommendation is to consider if the content of the tabs could be logically placed into one view and controller, then you probably don't want to use individual ionNavView components.

We will start by adding the ionNavView components into our tabs. We will have to give each one a name so they can be identified later. You can only have one ionNavView that is not named in your Ionic app, and the unnamed ionNavView is always the default view. Each tab will also be given a ui-sref attribute which will turn the tab icons into buttons to navigate between tabs. This section will not look drastically different when you preview it, but as you see in figure 5.4 it will now show a title in the header bar for the active tab.

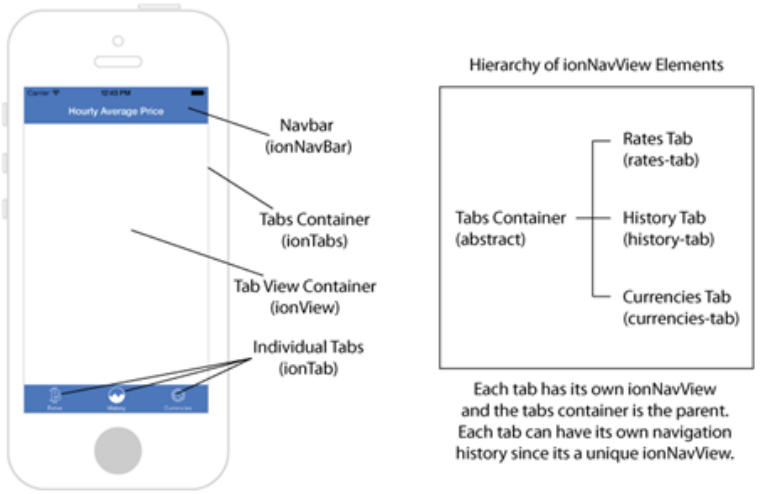


Figure 5.4 – Tabs with individual views, showing the title as you change tabs

Open the `www/views/tabs/tabs.html` template file and update to what you see in listing 5.4. Updates have been bolded for you.

Listing 5.4 – Tabs template with individual views (www/views/tabs/tabs.html)

```
<ion-tabs class="tabs-icon-top tabs-positive">
  <ion-tab title="Rates" icon-on="ion-social-bitcoin" icon-off="ion-social-bitcoin-
```

```
      outline" ui-sref="tabs.rates"> #1
    <ion-nav-view name="rates-tab"></ion-nav-view> #2
  </ion-tab>
  <ion-tab title="History" icon-on="ion-ios7-analytics" icon-off="ion-ios7-analytics-
      outline" ui-sref="tabs.history">
    <ion-nav-view name="history-tab"></ion-nav-view>
  </ion-tab>
  <ion-tab title="Currencies" icon-on="ion-ios7-cog" icon-off="ion-ios7-cog-outline"
      ui-sref="tabs.currencies">
    <ion-nav-view name="currencies-tab"></ion-nav-view>
  </ion-tab>
</ion-tabs>
```

#1 Add ui-sref to change the view on tab selection
#2 Define and name ionNavView for each tab

This adds 3 new ionNavView components with a different name. The ui-sref attributes act like a normal href attribute to link to a particular state based on the name, so instead of having a URL we have a state name. Even though only one of these three views will be visible, all three will be part of the same parent `tabs` view.

Now we need to add routes to our config that will support these new views. Ui-router has a feature called nested states, which allows you to declare states with a hierarchy. In this case, the tabs route that displays the tabs is like the root state, and each tab is a child state underneath it. This is helpful when you need to logically organize states and helps the Ionic navigation components understand your app's navigational structure. We need to update our app config with the new states and modify the tabs state as well. Listing 5.5 has the updated states configuration with updates bolded.

Listing 5.5 – App config with tab child states (www/js/app.js)

```
.config(function ($stateProvider, $urlRouterProvider) {
  $stateProvider
    .state('tabs', {
      url: '/tabs',
      abstract: true, #1
      templateUrl: 'views/tabs/tabs.html'
    })
    .state('tabs.rates', { #2
      url: '/rates', #3
      views: { #4
        'rates-tab': { #4
          templateUrl: 'views/rates/rates.html' #4
        } #4
      } #4
    })
    .state('tabs.history', { #5
      url: '/history', #5
      views: { #5
        'history-tab': { #5
          templateUrl: 'views/history/history.html' #5
        } #5
      } #5
    }) #5
    .state('tabs.currencies', { #6
      url: '/currencies', #6
```

```
      views: { #6
        'currencies-tab': { #6
          templateUrl: 'views/currencies/currencies.html' #6
        } #6
      } #6
    }); #6
  $urlRouterProvider.otherwise('/tabs/rates'); #7
})
```

#1 Update tabs state to be abstract, since we always want to use a child
#2 Declare the tabs.rates state, using dot notation for parent.child relationship
#3 Declare the url for the route, since it is a child route it appends this to url of parent
#4 Rates view targets the view with this name and pass a template for that view
#5 History view declaration
#6 Currencies view declaration
#7 Update default route to rates view

There are a few things going on here that are new in the states configuration. The tabs route (#1) now has the abstract: true property set, which makes it possible to declare it as a parent but does not allow it to be an active state.

When you look at the rates state, it has the name declared with tabs.rates (#2). This is to indicate the parent and child relationship they have. The URL is also declared here (#3), but take note that when you have a parent/child relationship, the URL is actually appended to the end of the parent URL. The rates view URL is actually found at `/tabs/rates` and not just `/rates`. Lastly, we declare subviews of the state (#4). The view must be named the same as the name you gave to the ionNavView earlier, in this case rates-tab. The app now knows that when the rates tab is active, it should inject the specified template into that view. Later we will declare other view properties such as controllers, when we are ready.

Lastly, the code updates the default route from /tabs to /tabs/rates (#7). This is because in our tabs you always want to be on one of the tabs, so the tabs container state is abstract. If you attempt to go to the tabs route (/tabs) it will now redirect you to the default rates view.

The last task in this section is to add the basic templates for each of our three tabs. We've already declared them in our states above using the `templateUrl` property in the view. The next three listings will contain a simple template that will contain a view and title for each tab.

Listing 5.6 – Rates tab basic template (www/views/rates/rates.html)

```
<ion-view title="Current Rates">
  <ion-content>
  </ion-content>
</ion-view>
```

Listing 5.7 – History tab basic template (www/views/history/history.html)

```
<ion-view title="Hourly Average Price">
  <ion-content>
  </ion-content>
</ion-view>
```

Listing 5.8 – Currencies tab basic template (www/views/currencies/currencies.html)

```
<ion-view title="Currencies">
  <ion-content>
  </ion-content>
</ion-view>
```

These templates are blank for the moment, but we will update each one individually in the following sections. If you preview the app now in your browser, you will be able to see the title changing as you change tabs in the bottom. This finishes what we need to do with tabs so let's get to work on creating the first tab and showing the current Bitcoin rates.

5.4 Loading and displaying current Bitcoin rates

Our app is all about showing information about Bitcoin, and the first tab is about showing the current market price for Bitcoin in different currencies. We will use a free service from Bitcoin Average (https://bitcoinaverage.com) which provides near real time rates and historical rates as well. It does this by averaging current market rates for Bitcoin across multiple exchanges, and the exchanges vary by currency. If you are following along using Git, you can check out the code for this step.

```
$ git checkout –f step4
```

In this section we're going to wire up the loading of the live data and display it in our tab. In figure 5.5 you can see the result of our work from this section. The date will appear with the last updated time for the results and the list of currencies will display the current prices and trend.

Figure 5.5 – Rates tab with data loading for current Bitcoin prices shown in a list component

To help facilitate our list of currencies, we are going first create a Currencies service. It is going to be very simple, just an array of the supported currencies for our app, but since it will be a service we will be able to reuse it in multiple parts of our app.

Open up the www/js/app.js file and add the code from listing 5.9.

Listing 5.9 – Currencies data service (www/js/app.js)

```
.factory('Currencies', function () { #1
  return [
    { code: 'AUD', text: 'Australian Dollar', selected: true }, #2
    { code: 'BRL', text: 'Brazilian Real', selected: false },
    { code: 'CAD', text: 'Canadian Dollar', selected: true },
    { code: 'CHF', text: 'Swiss Franc', selected: false },
    { code: 'CNY', text: 'Chinese Yuan', selected: true},
    { code: 'EUR', text: 'Euro', selected: true },
    { code: 'GBP', text: 'British Pound Sterling', selected: true },
    { code: 'IDR', text: 'Indonesian Rupiah', selected: false },
    { code: 'ILS', text: 'Israeli New Sheqel', selected: false },
    { code: 'MXN', text: 'Mexican Peso', selected: true },
    { code: 'NOK', text: 'Norwegian Krone', selected: false },
    { code: 'NZD', text: 'New Zealand Dollar', selected: false },
    { code: 'PLN', text: 'Polish Zloty', selected: false },
    { code: 'RON', text: 'Romanian Leu', selected: false },
    { code: 'RUB', text: 'Russian Ruble', selected: true },
    { code: 'SEK', text: 'Swedish Krona', selected: false },
```

```
    { code: 'SGD', text: 'Singapore Dollar', selected: false },
    { code: 'USD', text: 'United States Dollar', selected: true },
    { code: 'ZAR', text: 'South African Rand', selected: false }
  ];
});
```

#1 Register a service using Angular's factory method
#2 Create an array of currencies and set default selected state for each

This array contains a list of objects, with each object containing information about the currency. The code is the standard code for the currency, the text is the name of the currency, and the selected property is used to determine if that currency should be shown or not in the list. We'll make that configurable later, but by default some are set to false to disable them. Now that we've created and registered this service, we'll be able to use it anywhere in our app.

The first place we'll use the Currencies service will be in a controller for the rates view. This controller will take care of loading the current rates from Bitcoin Average's API. Once it is loaded, it will attach the data onto the Currencies service and that data will be made available on the scope. Listing 5.10 has the rates controller for `www/views/rates/rates.js`.

Listing 5.10 – Rates tab controller (www/views/rates/rates.js)

```
angular.module('App')
.controller('RatesCtrl', function ($scope, $http, Currencies) { #1

  $scope.currencies = Currencies; #2

  $scope.load = function () { #3
    $http.get('https://api.bitcoinaverage.com/ticker/all').success(function (tickers)
      { #4
      angular.forEach($scope.currencies, function (currency) { #5
        currency.ticker = tickers[currency.code]; #5
        currency.ticker.timestamp = new Date(currency.ticker.timestamp); #6
      });
    });
  };

  $scope.load(); #7
});
```

#1 Declare the RatesCtrl and inject the services used
#2 Immediately set the data from the Currencies service on the scope
#3 Scope method to load the data that can be called on demand
#4 Make the HTTP request to Bitcoin Average for current rates
#5 Loop over the list of currencies, and store the ticker data on the Currencies service
#6 Convert the timestamp from the response to a valid JavaScript date object
#7 Trigger a load when the controller is first loaded

This controller takes care of loading the data when the load method is called, using the $http service. The Currencies service is injected and stored on the scope, which our view will use to display all of the data. We are also storing the current rates on the Currencies service, which will come in handy later. This is a single data object that we will pass around and use

multiple times in other places. While you could use other techniques for sharing data, this approach works well for this particular situation.

Alright, so we have the ability to load our data, but now we'd like to display it on the screen. Let's update our template to loop over the currencies and display the data loaded by the controller. Open the www/views/rates/rates.html file and update it as you see in listing 5.11.

Listing 5.11 – Rates tab template with currency data (www/views/rates/rates.html)

```
<ion-view title="Current Rates">
  <ion-content>
    <ion-list>
      <ion-item ng-repeat="currency in currencies | filter:{selected:true}"> #1
        {{currency.code}} - {{currency.text}}
        <span class="price" ng-if="currency.ticker.last ==
       currency.ticker['24h_avg']"> #2
          {{currency.ticker.last || '0.00'}}<br />0.00 #2
        </span> #2
        <span class="price negative" ng-if="currency.ticker.last <
       currency.ticker['24h_avg']"> #3
          {{currency.ticker.last}}<br /><span class="icon ion-arrow-down-b"></span>
       {{currency.ticker['24h_avg'] - currency.ticker.last | number:2}} #3
        </span> #3
        <span class="price positive" ng-if="currency.ticker.last >
       currency.ticker['24h_avg']"> #4
          {{currency.ticker.last}}<br /><span class="icon ion-arrow-up-b"></span>
       {{currency.ticker.last - currency.ticker['24h_avg'] | number:2}} #4
        </span> #4
      </ion-item>
    </ion-list>
  </ion-content>
  <ion-footer-bar class="bar-dark"> #5
    <h1 class="title">Updated {{currencies[0].ticker.timestamp |
      date:'mediumTime'}}</h1> #5
  </ion-footer-bar> #5
</ion-view>
```

#1 ngRepeat to loop over currencies, and filter out any that are not active
#2 Price box shown when current price equal to 24h average
#3 Price box shown when current price below 24h average
#4 Price box shown when current price above 24h average
#5 ionFooterBar to keep a footer bar with last time data was loaded

This template has quite a bit going on, so let's start from the top. We've used the list component here, and then used ngRepeat to create a list item for each currency (#1). However, in the ngRepeat there is a filter, unfortunately named filter, which removes any of the currencies that have the `selected` property set to `false`. Later, we will make a configuration view that allows you to toggle currencies on or off, so this property is already filtering by the default settings in the Currencies service.

Inside of each of the items we are binding some text, and then there are 3 span elements with ngIf on them (#2, #3, #4). These are for displaying the current price and the trend compared to the past 24h average. There are three possible situations, the price is equal,

higher, or lower than the 24h average. These three spans account for those, and there are several expressions used to calculate the change in price and to determine which of the three situations to display the correct price formatting.

After we finish the list, you can see the ionFooterBar (#5) is placed after the end of the ionContent. The two components are aware of one another and aware of the tabs, so the footer is positioned above the tabs automatically and the content area is also sized based on the footer and tab bars at the bottom. This is important so the scrolling area is the correct size, but it is automatically handled for you by Ionic when you use these directives together.

We have to add some CSS to make our price boxes look correct, since Ionic doesn't have a component built for this exact purpose. The CSS is in listing 5.12 and you can place it in the `www/css/styles.css` file.

Listing 5.12 – Price box styling (www/views/css/styles.css)

```
.item .price { #1
  font-weight: bold; #1
  font-size: 13px; #1
  color: #fff; #1
  position: absolute; #1
  background: #666; #1
  right: 15px; #1
  height: 42px; #1
  top: 5px; #1
  width: 80px; #1
  text-align: center; #1
  padding: 6px; #1
  line-height: 1.2em; #1
}#1
.item .price.positive { #2
  background: #66cc33; #2
} #2
.item .price.negative { #3
  background: #ef4e3a; #3
} #3
```

#1 CSS rules for all price boxes
#2 CSS to change background color for positive change
#3 CSS to change background color for negative change

This CSS is modeled somewhat on the badges from Ionic, but the badges are not able to handle multiple lines. This is all of the custom CSS we will use for this app, so I have left it in the general styles.css file.

Ok we are almost done, we just need to add the controller to our state declaration and include the JavaScript file into our index.html. Open up the www/index.html file and add the script tag for our controller before the closing </head> tag after all other JavaScript files that are declared.

```
<script src="views/rates/rates.js"></script>
```

Lastly, we need to add the controller to our state, so open up www/js/app.js and modify the state as you see bolded below to add the controller for the rates tab view.

```
.state('tabs.rates', {
  url: '/rates',
  views: {
    'rates-tab': {
      templateUrl: 'views/rates/rates.html',
      controller: 'RatesCtrl'
    }
  }
})
```

Now if you reload the app in your browser, you should see the current rates loading for the currencies in the list. We've covered a lot in this section, but we can still make this experience better. In the next section we're going to introduce a new view to view the full details for a given currency instead of just the current price and trend.

5.5 Display a currency's details in the same tab view

The current rates are great, but markets have more information than we've shown so far. We want our app users to be able to see all of the available data, which includes the current ask, bid, and trade volume values. We can do this by creating a new view that the rates tab can navigate to, and use the back button on that tab only to go back to the main list. You can see the detail view in figure 5.6. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step5
```

Figure 5.6 – Detail view with back button, shows details about a currency while still on rates tab

You can see the rates tab is still active even though we will introduce another view, allowing this tab to have two levels of navigation with the back button to take you back to the main rates view. If you navigate to another tab and return, then the detail view will still be active with the back button to return to the rates. This allows us to remember the current state of a tab, which provides a better user experience.

Let's start by creating the controller for the detail view. It does not have to load any data itself, it just uses the Currencies service to display data that was already loaded in the rates view. Listing 5.13 has the controller that should go into `www/views/detail/detail.js`.

Listing 5.13 – Detail controller (www/views/detail/detail.js)

```
angular.module('App')
.controller('DetailCtrl', function ($scope, $stateParams, $state, Currencies) { #1

  angular.forEach(Currencies, function (currency) { #2
    if (currency.code === $stateParams.currency) { #2
      $scope.currency = currency; #2
    } #2
  }); #2

  if (angular.isUndefined($scope.currency.ticker)) { #3
    $state.go('tabs.rates'); #3
  } #3
```

```
});
```

#1 Register a controller and inject services
#2 Loop over each currency to find the requested currency, and store it on scope
#3 If the currency and ticker data isn't set, go back to rates view

When we declare this state, we will add a parameter called currency, and the controller uses the $stateParams service to access the value of that parameter. We'll see how that is passed to the state shortly. Once we know the currency, we loop over each of the currencies until the code matches and set the currency model on the scope for the template. Lastly, we check if the currency model is valid, and if not we go back to the rates view. Since this tab does not load data itself, if you refreshed the browser on the detail view it would have nothing to display and this will redirect to the rates view instead of showing a blank detail view.

Now let's get the template added for the detail view. The data needs to be displayed and we'll use the list and badges to show the values. At the bottom there will be a link to the historical data for that currency, which will link to another tab. Let's create the new template and place it in www/views/detail/detail.html with code from listing 5.14.

Listing 5.14 – Detail template (www/views/detail/detail.html)

```
<ion-view title="Detail for {{currency.code}}"> #1
  <ion-content>
    <ion-list>
      <ion-item>Last <span class="badge badge-
      stable">{{currency.ticker.last}}</span></ion-item> #2
      <ion-item>Ask <span class="badge badge-
      balanced">{{currency.ticker.ask}}</span></ion-item>
      <ion-item>Bid <span class="badge badge-
      assertive">{{currency.ticker.bid}}</span></ion-item>
      <ion-item>24h Average <span class="badge badge-
      dark">{{currency.ticker['24h_avg']}}</span></ion-item>
      <ion-item>24h Volume <span class="badge badge-stable icon ion-social-bitcoin">
      {{currency.ticker.total_vol | number:2}}</span></ion-item>
      <ion-item ui-sref="tabs.history({currency: currency.code})" class="item-icon-
      right">View History <span class="icon ion-arrow-right-b"></span></ion-item> #3
    </ion-list>
  </ion-content>
  <ion-footer-bar class="bar-dark">
    <h1 class="title">Updated {{currency.ticker.timestamp | date:'mediumTime'}}</h1>
  </ion-footer-bar>
</ion-view>
```

#1 Bind the currency code into the view title
#2 Display each of the values in a badge, which are floated right in a list
#3 Add link to view history, which links to the tabs.history state and passes a currency code param

This template has a list of the values we want to display, and uses the badges. Badges are used simply by adding an element with the badge class, and a badge-[color] preset class. The same color name guidelines apply, such as badge-assertive. We bind the values and also add a small Bitcoin icon for the volume value.

The last list item has the ui-sref attribute, just like on our tabs. However here we are using it like a function and passing an object, which is the way you can pass parameters to another state. We'll look at how the history tab uses this value in another section, but for the moment it will link to the history tab.

As usual, we need to include this new route in our state configuration and include the controller script in the index.html file. Add the following line to the index.html after the other scripts.

```
<script src="views/detail/detail.js"></script>
```

Now open up www/js/app.js and let's declare this new state. Add the following state definition into your config method from listing 5.15.

Listing 5.15 – Detail state definition (www/js/app.js)

```
.state('tabs.detail', {
  url: '/detail/:currency', #1
  views: {
    'rates-tab': { #2
      templateUrl: 'views/detail/detail.html', #3
      controller: 'DetailCtrl' #3
    }
  }
})
```

#1 The :currency indicates a parameter that will be the currency code
#2 We reuse the same rates tab view since this state is designed to be displayed there
#3 Declare the template and controller

The state declaration here is similar to the rates state, and reuses the same view. The :currency parameter will be set to a currency code, and passed to the state so it knows which currency to use. This value is made available to the $stateParams in the detail controller, which we already used in listing 5.13.

The last step is to make the list of items in the rates view link to the detail view for that currency. We need to update the rates template with two small changes, that are bolded in listing 5.16.

Listing 5.16 – Rate template update to link to detail view (www/views/rates/rates.html)

```
<ion-view title="Current Rates" hide-back-button="true"> #1
  <ion-content>
    <ion-list>
      <ion-item ng-repeat="currency in currencies | filter:{selected:true}" ui-
      sref="tabs.detail({currency: currency.code})"> #2
        {{currency.code}} - {{currency.text}}
… #3
```

#1 Add hide-back-button attribute so the back button doesn't appear on rates
#2 Add ui-sref and target the tabs.detail state, passing the currency code as a parameter
#3 The rest of the template remains the same

Here we've told the view should never display the back button. Since the current rates list is like the top level page, we don't want the user to be able to go back. They should select an item from the list to view instead.

Then we use the ui-sref attribute again and link to the tabs.detail state. We are passing the currency code as a parameter so the detail view knows which currency was selected.

We can now preview our app and when you click or tap on a currency, it will take you to the detail view. The back button will be visible for you to go back to the rates view. We're going to make one last improvement to the rates view and then we will build out the other two tabs.

5.6 Refresh the Bitcoin rates and display help

The rates are loading and we can view the detail, but currently there is no way to refresh the rates. Our app users will want to be able to get updated rates, and a common technique is to use the Ionic Refresher component that allows you to pull down on the screen and release to trigger a refresh of the data.

We also want to make sure that our users have a quick help panel that explains the information they are looking at. We will use the Ionic popover component to display this help information. Figure 5.7 has both the refresh component and popover component active states for you to preview. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step6
```

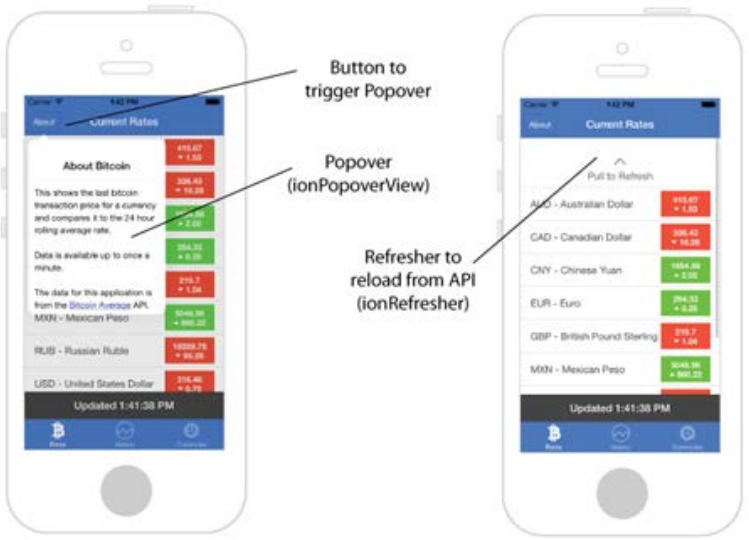


Figure 5.7 – Popover and pull to refresh components in action

5.6.1 ionRefresher: Pull to refresh the rates

Ionic's ionRefresher component allows any ionContent component to have a hidden panel that is displayed as the user pulls down on the content area, and if the user pulls far enough and releases it will call a function to reload data. When the reload has finished, the component will hide again.

We have to update both the rates template and controller to support the ionRefresher. First we need to add the ionRefresher component into our template, and in listing 5.17 you can add the new line as bolded to the template in www/views/rates/rates.js.

Listing 5.17 – Adding ionRefresher to rates template (www/views/rates/rates.js)

```
<ion-view title="Current Rates" hide-back-button="true">
  <ion-content>
    <ion-refresher on-refresh="load()" pulling-text="Pull to Refresh"></ion-refresher>
      #1
    <ion-list>
… #2
```

#1 ionRefresher component must be first inside of ionContent, and will call load method
#2 The rest of the template remains the same

This may seem deceptively simple, but this is all you have to do to add the component to the template. It will inject the hidden refresh component above the content, and when the user pulls the component will appear. It also shows an icon, which you can configure which icons are used, but we are using the default icon type. The pulling-text attribute lets us add a message to inform the user what this component will do.

When the refresh component is pulled far enough and released, the icon will change to a spinner and call the load() method declared using on-refresh. We already have a load method in our controller that handles the loading of the data, so the only thing we have left to do is tell the refresh component when the data has loaded. On its own, the component does not know when the data is done loading and will never hide. We have to update the load() method and broadcast an event that will tell the refresh component to complete.

Open up the rates controller in www/views/rates/rates.js and update the load method with the bolded portion in listing 5.18.

Listing 5.18 – Updating load method to close ionRefresher

```
$scope.load = function () {
  $http.get('https://api.bitcoinaverage.com/ticker/all').success(function (tickers) {
    angular.forEach($scope.currencies, function (currency) {
      currency.ticker = tickers[currency.code];
      currency.ticker.timestamp = new Date(currency.ticker.timestamp);
    });
  }).finally(function () { #1
    $scope.$broadcast('scroll.refreshComplete'); #2
  });
};
```

#1 Chain a finally method which fires after the HTTP request has completed, regardless of success or failure
#2 Broadcast the scroll.refreshComplete event so ionRefresher knows to close

We are using the finally method, which is part of the promises API, to broadcast the scroll.refreshComplete method regardless of success or failure of the HTTP request. We don't want the refresher to continue showing even if there was an error, so the finally method is able to execute no matter what. That is all we need to do to support the pull to refresh feature in our view. Planning ahead to make sure that you can reload data easily makes this component easy to implement.

5.6.2 $ionicPopover: Showing help in a popover

The Ionic popover component is typically used by having a button in the header that opens the popover. You are not limited to what you can put into a popover component, but the popover does take up only a portion of the screen. If you need to use the full screen, then you will need another component. In this case, we'll display some basic content about what is currently on the screen, and give credits to the source of the data.

The popover displays differently depending on what platform your app is running to mimic the style of the platform styling. You will likely want to avoid trying to change the styling of a popover container, because it will need to be verified on all platforms.

We'll start by adding a new template file with the contents of our popover. I like to think of the popover like a subview, where it loads a template into a container without creating a completely new view. I also suggest putting the template file inside of the view folder instead of in a new folder, so create a new file at www/views/rates/help-popover.html and insert the contents from listing 5.19.

Listing 5.19 – Popover template (www/views/rates/help-popover.html)

```
<ion-popover-view> #1
  <ion-header-bar> #2
    <h1 class="title">About Bitcoin</h1> #2
  </ion-header-bar> #2
  <ion-content> #3
    <div class="padding">This shows the last bitcoin transaction price for a currency
      and compares it to the 24 hour rolling average rate.</div> #3
    <div class="padding">Data is available up to once a minute.</div> #3
    <div class="padding">The data for this application is from the <a
      href="https://bitcoinaverage.com/api">Bitcoin Average</a> API.</div> #3
  </ion-content> #3
</ion-popover-view>
```

#1 Use the ionPopoverView to wrap the template, acts like ionView for popovers
#2 Use ionHeaderBar in the popover
#3 Use ionContent and add HTML content for popover

This template is wrapped in an ionPopoverView instead of ionView, because this is a specialized template just for popovers. Then we use the ionHeaderBar and ionContent components to wrap our content. Our content is simple HTML with text.

Now we need to register the popover so the view knows about it, and this is done in the controller. Much like we declare a state in the app config, we need to declare the popover in our controller. Since a popover is not meant to be a globally visible feature, we are able to

isolate it in one view to reduce overhead and complexity. Open the rates controller in www/views/rates/rates.js again and update it with listing 5.20.

Listing 5.20 – Registering popover with controller (www/views/rates/rates.js)

```
angular.module('App')
.controller('RatesCtrl', function ($scope, $http, $ionicPopover, Currencies) { #1

  $scope.currencies = Currencies;

  $ionicPopover.fromTemplateUrl('views/rates/help-popover.html', { #2
    scope: $scope, #2
  }).then(function (popover) { #3
    $scope.popover = popover; #3
  }); #3
  $scope.openHelp = function($event) { #4
    $scope.popover.show($event); #4
  }; #4
  $scope.$on('$destroy', function() { #5
    $scope.popover.remove(); #5
  }); #5
… #6
```

#1 Inject the $ionicPopover service
#2 Declare a popover from a template URL and assign the parent scope as scope
#3 When the template has loaded, assign the popover to the scope
#4 Scope method to open the popover, requires the $event to be passed
#5 Listen for the $destroy event, which is when the view is destroyed and clean up popover
#6 The rest of the controller remains the same

First we have to inject the $ionicPopover service. Then we use it to create a new popover from a template URL. The popover does create its own scope, but we connect the scopes by passing an object with {scope: $scope}. Often you will need this so the popover can access the parent scope. The then method executes when the template has loaded and assigns a new popover to the scope.popover property.

Now the popover has been setup, and we are able to use the popover.show($event) method to show the popover. We will need to add an ngClick to a button to trigger it, but the $event is required to be passed. The $event value is the event object from the click event, which contains the information about which element was clicked. The popover uses that information to calculate where on the page to put the popover. There is also a popover.hide() method which can programmatically hide the popover, or the user can tap on the area outside of the popover to close it.

Lastly, we are listening for the scope $destroy event, which fires when the current scope is unloaded from memory. To prevent memory leaks, we remove the popover from the application since we are no longer using it.

Why do some components need to be manually removed?

Most of the components in Ionic can be cleaned up automatically when they are no longer in use, which helps to free up memory and improve performance. Some components, namely modals and popovers, require the app to remove the component when the scope is destroyed.

The $destroy method fires when the current scope has been deleted from memory. Anything that exists in that scope is removed at the same time, but popovers and modals both create isolated scopes that persist. Because of this architecture, there isn't an automatic way to remove the modal or popover from memory.

If you forget to do this in your app, it probably won't cause your app to become very slow. It depends on the complexity and memory usage of the popover or modal how much of an impact it might have. It might not have a noticeable impact on most apps if it is forgotten, but it is best to remove them.

Now let's add the button that will trigger the popover. Open up the rates template (www/views/rates/rates.html) one last time and add the bolded code in listing 5.21.

Listing 5.21 – Adding button to trigger popover (www/views/rates/rates.html)

```
<ion-view title="Current Rates" hide-back-button="true">
  <ion-nav-buttons side="primary"> #1
    <button class="button" ng-click="openHelp($event)">About</button> #2
  </ion-nav-buttons>
  <ion-content>
…#3
```

#1 ionNavButtons allows us to declare buttons in the top nav bar area
#2 Add a button, and use ngClick to call the openHelp while passing the $event
#3 The rest of the template remains the same

Now our new button will call the function to open the popover, and the popover will position itself to be under the button. The $event value here is a special Angular feature available for ngClick and other event directives that passes the event object along. The nav buttons will appear on the primary side, which may vary from platform to platform if that is the right or left.

That completes the rates view, we've added a popover for help information and a pull to refresh feature for updating the rates in our list. Next we will tackle the history tab which will load data and chart the historical price for the past month.

5.7 Charting historical data

Our app users will want to be able to see how the Bitcoin price has been trending over the past month for a given currency. We will use the popular Highcharts charting library, along with an Angular directive for Highcharts called highcharts-ng. This is not meant to be a primer on how to use Highcharts, but you can view the documentation at http://highcharts.com. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step7
```

We will be loading data again from Bitcoin Average's API, but this time the data will come not as JSON but as CSV (comma separated value) data. This particular API does not send back JSON data, so we will have to parse and format the data into a format that Highcharts can understand.

You can see the result of this section in figure 5.8. There is the chart as well as a box above that has the name of the currency. This is a select box that allows us to change the currency for the chart.

Figure 5.8 – History tab showing a chart of average price over the past month, with box to change currency in the chart

5.7.1 Setting up third party libraries

Our app is going to use some third party libraries, so we need to download a copy of them and set them up in our app. We are going to use Bower, which is a utility for downloading libraries into your project. Ionic already has it setup to use, but you do need to have Bower installed on your machine first. Just like Ionic itself, it is installed using NPM. You can learn more about Bower at http://bower.io.

```
$ npm install -g bower
```

This will take a moment to install Bower, but when it completes it should be ready to run. Then we need to install two libraries, the Highcharts charting library and the Angular wrapper

for Highcharts called highcharts-ng. We can have Bower download and put the recent copy into our project using the following command.

```
$ bower install --save highcharts-release#4.0.4 highcharts-ng#0.0.7
```

Once Bower has grabbed a copy of these two libraries, it might also install some additional files from Ionic. With this command we've chosen to install a specific version of each library, just so we can be sure the example in this book works as expected. They are downloaded and stored in the www/lib directory. Now we need to include the necessary script tags in our index.html. The first two are for Highcharts and the third is the Highcharts Angular wrapper.

```
<script src="lib/highcharts-release/adapters/standalone-framework.js"></script>
<script src="lib/highcharts-release/highcharts.js"></script>
<script src="lib/highcharts-ng/dist/highcharts-ng.js"></script>
```

The last step is to declare the highcharts-ng module as a project dependency, so we can use it. Open up www/js/app.js and add it as a new dependency.

```
angular.module('App', ['ionic', 'highcharts-ng'])
```

Now we are setup with the third party scripts that we need, so let's move on to building up our template for the history tab.

5.7.2 History tab template using Highchart and a select box to toggle currency

We created a blank template for the history tab before, so we need to update it to include the select box component for our currency selector, and setup the Highcharts component. We'll use an inset list with just one item to create the select box container.

Listing 5.22 – History template with chart (www/views/history/history.html)

```
<ion-view title="Hourly Average Price" hide-back-button="true"> #1
  <ion-content>
    <div class="list list-inset"> #2
      <label class="item item-input item-select"> #2
        <div class="input-label"> #2
          Currency #2
        </div> #2
        <select ng-change="changeCurrency()" ng-model="history.currency"> #3
          <option ng-repeat="currency in currencies | filter:{selected:true}"
       value="{{currency.code}}" ng-selected="history.currency ==
       currency.code">{{currency.code}} - {{currency.text}}</option> #4
        </select>
      </label>
    </div>
    <highchart config="chart"></highchart> #5
  </ion-content>
</ion-view>
```

#1 Hide the back button on this view
#2 Use the inset list to contain the select box
#3 Use a normal HTML select box, with ngChange and ngModel to track value changes
#4 Create an option for each active currency
#5 Highchart component which accepts a config attribute with a chart object

The select box component is based on the default HTML select box and is styled by Ionic to give it a mobile friendly appearance. When a select box is used on a mobile device, the platform takes over and provides the experience. There is little control apps have over this, but in our case we are happy to let the platform display the select box in a way that feels most native for that platform.

We again will use the Currencies service to display a list of the active currencies in the select box. The ngModel allows us to track the value of the select box. When the value changes, the ngChange triggers the changeCurrency() method, which will update the view to display the chart for that currency.

Lastly, the highchart directive used here takes a chart object, which we will declare in our controller. Based on the values of our chart object, the directive will work with Highcharts to render a chart based on the data we will load.

This is all we need in our template, but our controller has to do a bit of work to make everything behave properly.

5.7.3 History tab controller loads data and sets up chart

Our controller will need to handle setting up the chart, loading the chart data, and formatting it so the chart can use it. We will again turn to the $http service to load the data, and format the chart object according to the rules that Highcharts will understand. Since the data we are getting is not in the exact format, we'll be converting data before we can display it. The controller also will handle changing the currency and load the list of Currencies to use in the template.

Create a new controller at www/views/history/history.js and add the code from listing 5.23 into it. We'll break down the code carefully since there is a lot going on.

Listing 5.23 – History controller (www/views/history/history.js)

```
angular.module('App')
.controller('HistoryCtrl', function ($scope, $http, $state, $stateParams, Currencies)
      { #1

  $scope.history = { #2
    currency: $stateParams.currency || 'USD' #2
  }; #2
  $scope.currencies = Currencies; #3

  $scope.changeCurrency = function () { #4
    $state.go('tabs.history', { currency: $scope.history.currency }); #4
  }; #4

  $scope.chart = { #5
    options: { #5
      chart: { #5
        type: 'line' #5
      }, #5
      legend: { #5
        enabled: false #5
      } #5
    }, #5
    title: { #5
```

```
      text: null #5
    }, #5
    yAxis: { #5
      title: null #5
    }, #5
    xAxis: { #5
      type: 'datetime' #5
    }, #5
    series: [] #5
  };  #5

  $http.get('https://api.bitcoinaverage.com/history/' + $scope.history.currency +
      '/per_hour_monthly_sliding_window.csv').success(function (prices) { #6

    prices = prices.split(/\n/); #7
    var series = { #8
      data: [] #8
    }; #8

    angular.forEach(prices, function (price, index) { #9
      price = price.split(','); #10
      var date = new Date(price[0].replace(' ', 'T')).getTime(); #11
      var value = parseFloat(price[3]); #11
      if (date && value > 0) { #12
        series.data.push([date, value]); #12
      } #12
    });

    $scope.chart.series.push(series); #13
  });
});
```

#1 Create the controller and inject services
#2 Define the history model we set on the select box, defaulting to USD
#3 Store the list of currencies on the scope
#4 Function to handle changing the state after a new currency is selected
#5 Chart definition object that the highcharts directive turns into a chart
#6 Load the history information based on selected currency
#7 Split the prices string into an array of rows of prices
#8 Create a blank series to store all of the data in
#9 Loop over each row of the prices
#10 Split each row from a comma separated string to an array
#11 Parse and format the time and price values
#12 If the date and value are valid, add the point to the series
#13 Add the completed series of data to our chart

There seems to be a lot going on, but most of it is pretty straight forward. Let's start from the top. The first thing we do is set the history model and this contains the currency value from the $stateParams. If no currency was provided, we then default to USD. Then we store the currencies onto the scope for the template.

The changeCurrencies() method takes the value of the select box and updates the current state to use it. It calls the $state.go method, which is the programmatic equivalent to ui-sref in the template.

The rest of the controller is dedicated to the chart. We start by making a chart object, which is used by the highcharts-ng module and handles creating a chart for us. You can review the

documentation for highcharts-ng to get a full understanding of the object https://github.com/pablojim/highcharts-ng.

The last part is the loading and formatting of data. The $http service loads the price data in a CSV format, because that is all the API provides. This is problematic because JavaScript does not have a built way to handle CSV, but we can still parse it ourselves. The chart needs a series, which is a single set of data, and we create a blank series. Using the split method we break the CSV into JavaScript arrays that we can work with. The data also comes with some metadata that we don't want, so we filter it out and use only the data points. Then we add the data points to the series, and add the series to the chart. At this point, the line will appear with the price data.

Let's finish this tab by adding the controller script in our index.html file, and then updating the state definition to include the parameter and controller. In index.html, add the script tag for the history controller at the end of the existing scripts.

```
<script src="views/history/history.js"></script>
```

Then open up the www/js/app.js file and modify the history state to the following in listing 5.24. The updates are in bold.

Listing 5.24 – Updated state definition for history tab (www/js/app.js)

```
.state('tabs.history', {
  url: '/history?currency', #1
  views: {
    'history-tab': {
      templateUrl: 'views/history/history.html',
      controller: 'HistoryCtrl' #2
    }
  }
})
```

#1 Add the currency parameter for this state
#2 Declare the controller for this view

Now you can preview the app and view the history tab. The chart will load and you can change the selected currency to view another chart. We also already have support to link to the history tab in the detail view of the rates tab. If you go back to the rates tab and view the detail for a currency, select the View History link and it will take you to the history tab for that currency.

Our final task is to setup the currencies tab, which will allow us to toggle and reorder the currencies in the other tabs.

5.8 Currencies tab with list reordering and toggles

The last tab will take our list of currencies and let us change the order of the currencies and toggle currencies on or off for display. This is like a preferences screen so that a user is able to decide which currencies they care about and ignore the rest, or move their favorite currencies

up to the top. You can see the list of currencies with toggles and reordering in action in figure 5.9. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step8
```

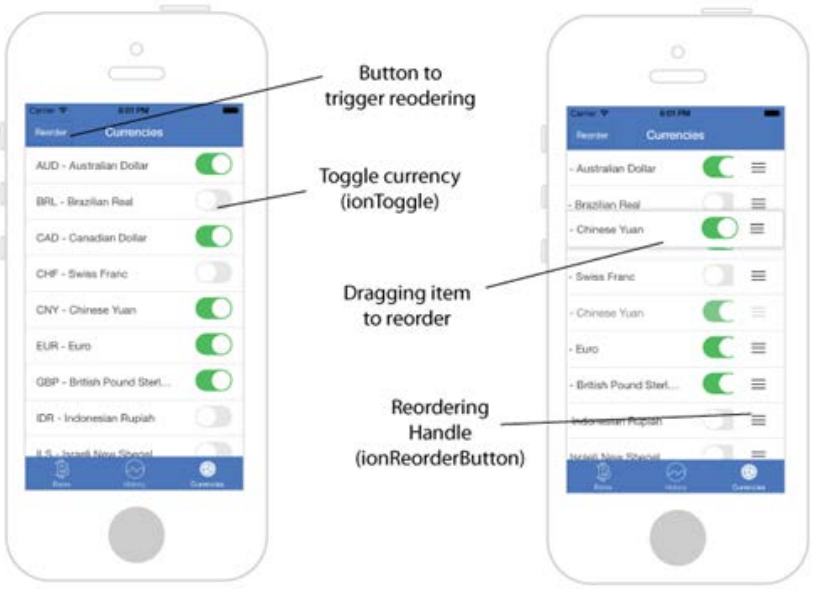


Figure 5.9 – Currencies tab with list of currencies to toggle on or off, and ability to reorder the list

5.8.1 ionReorderButton: Adding reordering to a list

Let's start by adding the template for the currencies tab, and add the reordering feature. Reordering can only work with the ionList directive. It works by setting a reordering state to true or false, and based on that value the reordering handles appear or hide. When they are activated, you can drag the item using the handle to a new position, and then your controller will handle updating the model to reflect the new ordering. Open up the currencies template at www/views/currencies/currencies.html and update it to reflect listing 5.25.

Listing 5.25 – Currencies template (www/views/currencies/currencies.html)

```
<ion-view title="Currencies">
  <ion-nav-buttons side="primary"> #1
    <button class="button" ng-click="state.reordering =
      !state.reordering">Reorder</button> #1
  </ion-nav-buttons> #1
  <ion-content>
    <ion-list show-reorder="state.reordering"> #2
      <ion-item ng-repeat="currency in currencies">
        {{currency.code}} - {{currency.text}}
        <ion-reorder-button class="ion-navicon" on-reorder="move(currency, $fromIndex,
      $toIndex)"></ion-reorder-button> #3
      </ion-item>
```

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

#1 Add button that toggles the state.reordering value
#2 Use show-reorder to declare the list can be reordered, and what model to use to activate
#3 ionReorderButton must be included and calls a method after moving using on-reorder

We've created a list of currencies that are able to be reordered. The ionList component uses the show-reorder attribute to evaluate if the ionReorderButton should be shown or not. These two work together to create the reorder functionality. The button in the navbar is used to toggle the state.reordering property, which will trigger the reordering to show or hide.

The on-reorder method allows us to write a method that handles what to do when the reordering is completed. It provides two special parameters, $fromIndex and $toIndex. This gives you the index values for the item in the array so you know the position to move it from and to. We will add this method in our controller next. Create a new file at www/views/currencies/currencies.js and insert the contents of listing 5.26 into it.

Listing 5.26 – Currencies controller (www/views/currencies/currencies.js)

```
angular.module('App')
.controller('CurrenciesCtrl', function ($scope, Currencies) { #1
  $scope.currencies = Currencies; #2
  $scope.state = { #3
    reordering: false #3
  }; #3

  $scope.$on('$stateChangeStart', function () { #4
    $scope.state.reordering = false; #4
  }); #4

  $scope.move = function(currency, fromIndex, toIndex) { #5
    $scope.currencies.splice(fromIndex, 1); #5
    $scope.currencies.splice(toIndex, 0, currency); #5
  }; #5
});
#1 Declare the controller and inject the services
#2 Attach currencies to the scope
#3 Declare the default reordering state value
#4 Listen for state changes, and turn off reordering when leaving tab
#5 Handle moving an item from one index value to another by splicing the item in the
      array
```

Our controller is fairly lean and starts by setting values on the scope. We listen for the $stateChangeStart event because we want to disable reordering anytime the currencies tab loses focus, and this even will fire anytime the tab changes. This is just a convenience for users, otherwise if the reordering was active when they leave the tab it will still be active when they return. The move method takes the item that is being moved along with the two index values for where it was and where it needs to go. Using splice, we remove it first from the array and then reinsert it at the new location.

At this point our list of currencies can be reordered, and once the items are reordered the other tabs will also reflect the new ordering. This is part of the power of using a shared service like our Currencies service. Changes made in one place are reflected across any state using the same service.

5.8.2 ionToggle: Adding toggles to list items

We also want to be able to toggle currencies on or off so we only have to see the items we care about. There is an ionToggle component you can use, but we are going to use the CSS version of the toggle because the ionToggle component doesn't work well with the ionReorderButton. The ionToggle component is just a helpful abstraction of the CSS version that we will be using, but does not provide any extra features.

Open up the currencies template once more, and we'll add the toggle component in for the final feature. In listing 5.27 you will see the additions to make for the ionItem component to include the toggle.

Listing 5.27 – Adding toggle to currencies list (www/views/currencies/currencies.html)

```
<ion-item class="item-toggle" ng-repeat="currency in currencies"> #1
  {{currency.code}} - {{currency.text}}
  <label class="toggle toggle-balanced"> #2
  <input type="checkbox" ng-model="currency.selected"> #3
    <div class="track"> #4
      <div class="handle"></div> #4
    </div> #4
  </label>
  <ion-reorder-button class="ion-navicon" on-reorder="move(currency, $fromIndex,
      $toIndex)"></ion-reorder-button>
</ion-item>
```

#1 Add the item-toggle class to get toggle styling
#2 Declare a label with the toggle class
#3 Use a checkbox input and give it a model for currency.selected
#4 Add elements needed for CSS to style a toggle icon

This toggle component uses the checkbox input to keep track of the value for the toggle. Checkboxes and togglers are both a Boolean value, so the CSS styling of a toggle overlays a traditional HTML checkbox. We use the model currency.selected for each currency, which we use to filter out items that are not enabled in the other tabs. As we toggle any item on or off, the other tabs are updated immediately to show or hide the currency. The power of the shared Currencies service is at work again.

You can now preview the app and everything should be complete. The app allows you to view the current rates for a currency and details about that currency, view a chart of the rates monthly history, and configure which of the currencies you wish to view and order them as you desire.

5.9 Chapter challenges

You now have a lot of components under your belt. To give yourself some challenges, you can attempt the following tasks to further improve your understanding and familiarity with these components.

- **Auto-refresh the rates.** Work on a technique to automatically refresh the rates once a minute. Angular has a useful $interval service that you might consider using.
- **Persist the currency settings.** There are techniques such as using localStorage or indexedDB that you can use to persist the currencies ordering and toggle states. Work on adding logic to manage the loading of currency from a cache before resetting the values to a default.
- **Chart more data.** The Bitcoin Average API provides more types of historical data, such as price since Bitcoin start. Add more configuration to the history tab that allows a user to change to different types of chart data. Review the API details on Bitcoin Average.
- **Improve the detail view.** The detail view is very basic and just lists information. Try to make a more compelling visual experience using more of Ionic's CSS components or by creating your own.

5.10 Summary

We've covered a lot in this chapter about Ionic components and leveraging the Highcharts library with data from the BitcoinAverage API.

- Tabs are a great way to provide a navigational structure in your app. Sometimes you need basic tabs and sometimes you need tabs with individual navigational histories like in our example app.
- Including third party scripts and Angular modules is easy to do, but each module has its own features that have to be learned individually.
- Lists have the ability to be reordered, have support for badges, and are able to include toggle components.
- Using a shared service like the Currencies service makes it possible to share data between views.

6

Using side menus for navigation

In this chapter you will learn how to:

- Use a side menu as the basis of your app for navigation
- Use action sheets, and popups to display options to users
- Use a modal to display related content
- Build more advanced scrolling interactions

During this chapter you will be building a weather app, and in the process we will showcase more components that Ionic has to offer. The base of the application navigation will be to use the side menu component. It will allow you to find and view weather conditions and forecasts, favorite locations, display sunrise and sunset data inside of a modal window, and use a paginated scrolling pane to view the weather information.

Throughout the chapter we will be looking at a number of Ionic's features and components. The side menu will be the basis for our app navigation, and we will use just a single left menu that can appear to navigate around the app. We will use the action sheet component to provide the user a list of options, such as to favorite a location. Using a modal, we will display the next year's chart of sunrise and sunset values. To make this chart perform better we will use the collection repeat feature of the Ionic lists, which reduces memory for large lists by rendering only the necessary items.

We will use two different services for loading data in this app. Forecast.io is a popular weather API service which provides current conditions and forecast data for a given geolocation based on latitude and longitude. In order to determine the locations, we will use Google's geolocation service to search for locations and their coordinates. Both are free, however you will need to register for an API key to use Forecast.io.

In figure 6.1 you can see many of the different views of this app. We will build them up piece by piece, but the bulk of the app will exist inside of the Weather view.

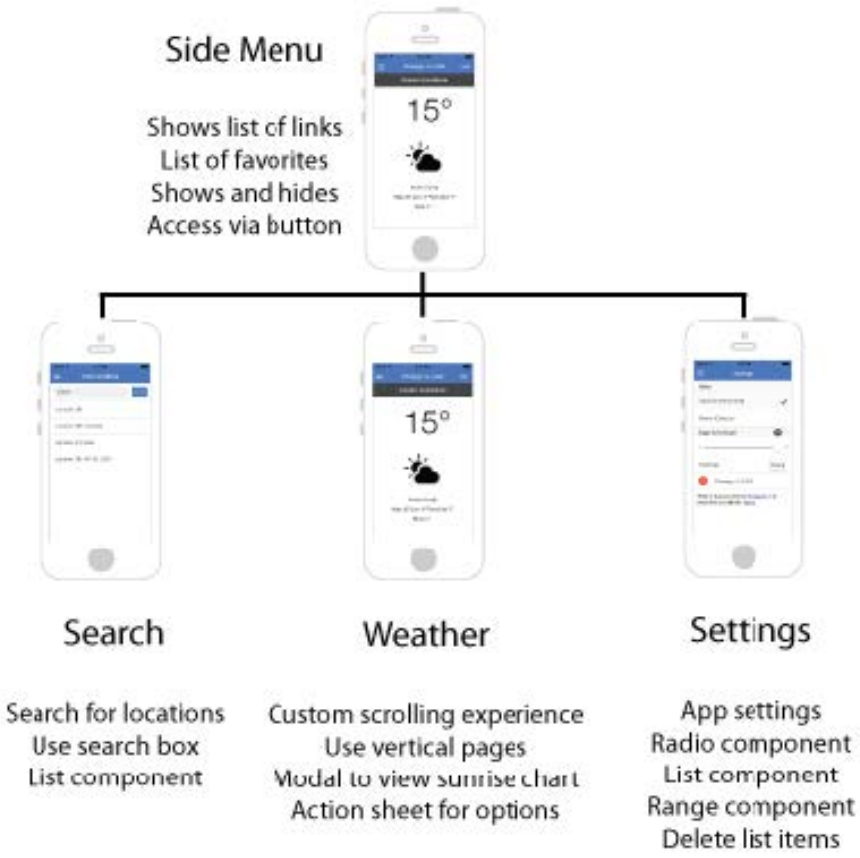


Figure 6.1 – The chapter example, side menu allows you to navigate between views

You can view the completed project at https://ionic-in-action-chapter6.herokuapp.com and the source code at https://github.com/ionic-in-action/chapter6.

6.1 Setup chapter project

You can follow along in this chapter by either creating a new Ionic app and adding the code from the listings in this chapter, or by cloning the finished app from the Ionic in Action GitHub repository and following along with each step. At the end start to serve the project so you can preview the app in a browser.

6.1.1 Create a new app and add code manually

To create a new project for our app using the Ionic CLI utility, open the command line and execute the following command. Remember, you can refer to chapter 2 if you need to refresh how projects are setup.

```
$ ionic start chapter6 https://github.com/ionic-in-action/starter
$ ionic serve
```

6.1.2 Clone the finished app and follow along

To checkout the finished app and use Git to follow along for each step, use the following command to clone the repository and checkout the first step.

```
$ git clone https://github.com/ionic-in-action/chapter6.git
$ cd chapter6
$ git checkout –f step1
$ ionic serve
```

6.2 Setting up the side menu and views

We've see how to build navigation yourself and how to use tabs, now we'll use side menus for primary navigation. Side menus are used frequently because they slide in and out of view on demand, allowing you to provide quick access to primary links but not use visual space unless it is required. A side menu can be opened on the right or left at once, and in our example we will use just the left side.

A side menu can be opened in three ways, depending on the implementation. By default, Ionic supports side menus opening by swiping to the side to pull the side menu open. You can also disable this in case you need to be able to use swipes for another purpose. You can use a button to open the side menu, usually placed in the top left corner of the screen. Lastly, you are able to toggle the menu programmatically using the side bar delegate service. If you are following along using Git, you can check out the code for this step.

```
$ git checkout –f step2
```

In our example for this chapter we'll use just one side menu, but you are able to use multiple side menus in a single app. There are many configuration options and ways to leverage the side menu, but they all leverage the same basic structure we'll use.

In this section we will setup the base application and navigation using side menus. This is done using the ionSideMenus components, and we will allow the side menu to appear from swiping to the right, or by using the toggle icon in the top left. We will also setup two blank routes that we will fill in later. You can see the side menu in action in figure 6.2, and this will be the result of our work in this section.

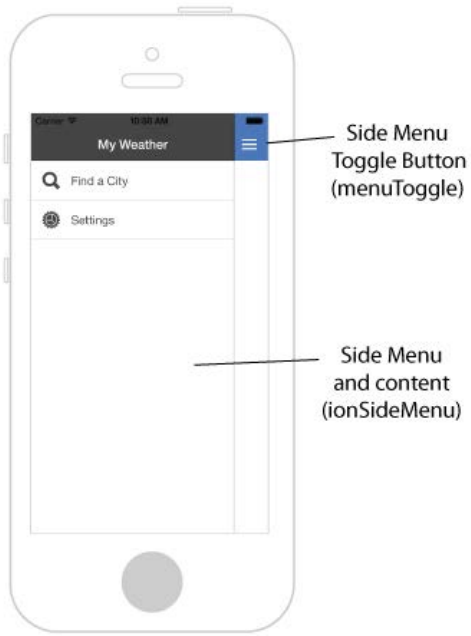


Figure 6.2 – Side menu in action, the left is the closed state and right is the opened state

To begin, we're going to modify the index.html for our app to setup our side menu. Listing 6.1 contains the side menu and content area.

Listing 6.1 – Side menu setup (www/index.html)

```
<body ng-app="App">
  <ion-side-menus> #1
    <ion-side-menu-content> #2
      <ion-nav-bar class="bar-positive"> #3
        <ion-nav-buttons side="left"> #3
          <button class="button button-clear" menu-toggle="left"><span class="icon
     ion-navicon"></span></button> #3
        </ion-nav-buttons> #3
      </ion-nav-bar> #3
      <ion-nav-view></ion-nav-view> #3
    </ion-side-menu-content>
    <ion-side-menu side="left"> #4
      <ion-header-bar class="bar-dark"> #5
        <h1 class="title">My Weather</h1> #5
      </ion-header-bar> #5
      <ion-content> #6
        <ion-list> #6
          <ion-item class="item-icon-left" ui-sref="search" menu-close><span
     class="icon ion-search"></span> Find a City</ion-item> #6
          <ion-item class="item-icon-left" ui-sref="settings" menu-close><span
     class="icon ion-ios7-cog"></span> Settings</ion-item> #6
        </ion-list> #6
```

```
      </ion-content> #6
    </ion-side-menu>
  </ion-side-menus>
</body>
```

#1 Declare the ionSideMenus container to wrap side menu and content areas
#2 Use ionSideMenuContent to hold the main center content
#3 Using navigation components inside of side menu content area, with toggle icon
#4 Declare a side menu, assigning it to the left side
#5 Using a new header for the side menu
#6 ionContent is used with a list of links for navigation

Side menus are easy to declare, as they only require using the ionSideMenus, ionSideMenuContent, and ionSideMenu directives in the markup. No JavaScript is required to setup the side menu. We first wrap the entire content area with ionSideMenus, which takes care of setting up the functionality based on the other directives that are declared. Without it, the side menu would not function. Inside of the ionSideMenus we add two child elements, ionSideMenuContent and ionSideMenu. This declares we have one side menu, which we define to be on the left, and the main content area. You can only declare one ionSideMenuContent element for each side menu, but you can declare up to two ionSideMenu elements for the right and/or left.

Inside of the content area, we have declared the same navigational directives we've used in the past. This way, our side menu acts like a global base that contains our navigation view container. I think this structure makes the most sense because your side menu typically is used for global navigation in your app.

If you look at the ionNavButtons, you'll see we've declared a single button with `menu-toggle="left"`. The menuToggle directive is used to take care of toggling the side menu open or closed when the button is activated. Likewise, in the side menu item list you see the `menu-close` directive on the navigation links. The menuClose directive will close any open side menu when the item is activated. When you tap on "Find a City" it will close the left menu automatically, otherwise the side menu would remain open even while the navigation area updated with the new content.

You should think of each ionSideMenu and the ionSideMenuContent like their own views. In the side menu, I've used a header bar and content area to wrap the navigation list, otherwise the content area would not calculate the correct size and location of elements.

The side menu can contain any content you want, but a list of navigation links is the most common use case for the side menu. You might also use a right side menu to provide additional search filters or even secondary navigation.

We've done everything for the side menu in listing 6.1. You can review the side menu documentation to see some other features and get details about the features of the delegate service in case you need to have programmatic control over the side menu.

The links declared in the side menu currently will not work until we declare those routes, but you can toggle the side menu open and closed or swipe to pull it open. We'll start by first setting up the search view, which will allow us to find locations and their coordinates.

6.3 Searching for locations

When the app first starts, a user will need to be able to configure the locations that they would like to view. Using Google's Geolocation API, we can search for locations by any type of input text, such as a zip code, city name, and even more specific locations such as a particular address. We'll create a new view that allows us to search and view a list of results from this API. You can see the search in action in figure 6.3. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step3
```

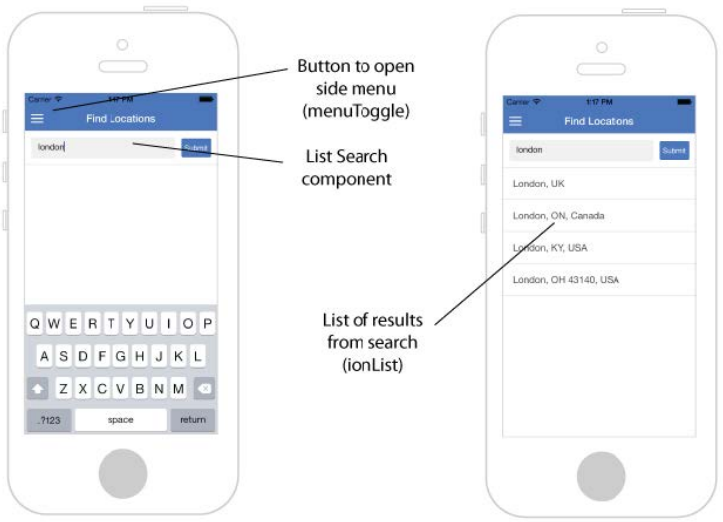


Figure 6.3 – Search view in action, with keyboard and results

To accomplish this, we'll need to register a new state with our state provider and define the template and controller. Our template will contain a search field and a button, while our controller will handle the API request to get the list of results. By now you should be familiar with declaring a new state, so let's start with that inside of our www/js/app.js file found in listing 6.2.

Listing 6.2 – Declare the search state (www/js/app.js)

```
angular.module('App', ['ionic'])
.config(function ($stateProvider, $urlRouterProvider) { #1

  $stateProvider
    .state('search', { #2
      url: '/search', #2
      controller: 'SearchCtrl', #2
      templateUrl: 'views/search/search.html' #2
    });
```

```
  $urlRouterProvider.otherwise('/search'); #3
})
```

#1 Add a config() method for our app
#2 Declare the search state
#3 Use the search as the default view

Since this is the first state, we need to add the config method and then inject the $stateProvider and $urlRouterProvider services. We declare the search state, and set it to the default route. Now we need to add our template and controller.

The template for the search view is in listing 6.3, and contains a search box and a list component to display the list of results. Create a new file at www/views/search/search.html and add the contents in the listing.

Listing 6.3 – Template for search (www/views/search/search.html)

```
<ion-view title="Find Locations">
  <ion-content>
    <div class="list">
      <div class="item item-input-inset"> #1
        <label class="item-input-wrapper"> #1
          <input type="search" ng-model="model.term" placeholder="Search for a
      location"> #1
        </label> #1
        <button class="button button-small button-positive" ng-
      click="search()">Submit</button> #1
      </div>
      <div class="item" ng-repeat="result in results" ui-sref="weather({city:
      result.formatted_address, lat: result.geometry.location.lat, lng:
      result.geometry.location.lng})">{{result.formatted_address}}</div> #2
    </div>
  </ion-content>
</ion-view>
```

#1 Search list item with ngModel for the search box and a button
#2 Repeat over list of results when available to display address, and link to weather view

Here we have a basic template with a list. The first list item is the search box, and then if any results exist they will be displayed below it. The style of this box is to have an inset box, to give it a little different visual appearance. I've also declared the input to be the search type, because it will modify the display of the keyboard for searching on a device.

We haven't declared the weather state yet, but you can see I've already added the ui-sref for it. In this case, we will be passing the city, latitude, and longitude values from the result.

To power this template, we need the controller. Create a new file at www/views/search/search.js and add the code from listing 6.4.

Listing 6.4 – Search controller (www/views/search/search.js)

```
angular.module('App')
.controller('SearchCtrl', function ($scope, $http) {
  $scope.model = {term: ''}; #1
```

```
  $scope.search = function () { #2
    $http.get('http://maps.googleapis.com/maps/api/geocode/json', {params: {address:
      $scope.model.term}}).success(function (response) { #2
      $scope.results = response.results; #2
    }); #2
  }; #2
});
```

#1 Define the search term model
#2 Method to handle searching from geocoding API using the term and storing on scope

In our controller we define the default model, which will be reset every time the view is loaded. Then the search method is called when the button is tapped, and makes the HTTP request to the Geocoding API. The response gets stored on the scope, which will update the view with the list when it is available. The results can be a little inconsistent with the information provided, but that is due to the way the service sends data back.

Our search view is now complete. Next we will be building out the settings view and a few custom services that we'll use to store and share data.

6.4 Adding Settings view and data services

Our app needs to have some configuration options, particularly to allow users to select what type of units they wish to see (such as temperatures in Fahrenheit or Celsius). It will then allow users to select how many days to view for the forecast. Lastly it will allow users to manage their list of favorite locations by deleting items in the list. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step4
```

We will need to add a new state with a controller and template for our settings view. Then to manage our app, we will need two services that can be used to shared data and methods between views. Lastly, we will also update the side menu to include a list of the favorite locations for quick access. Review figure 6.4 to see the finished view in action.

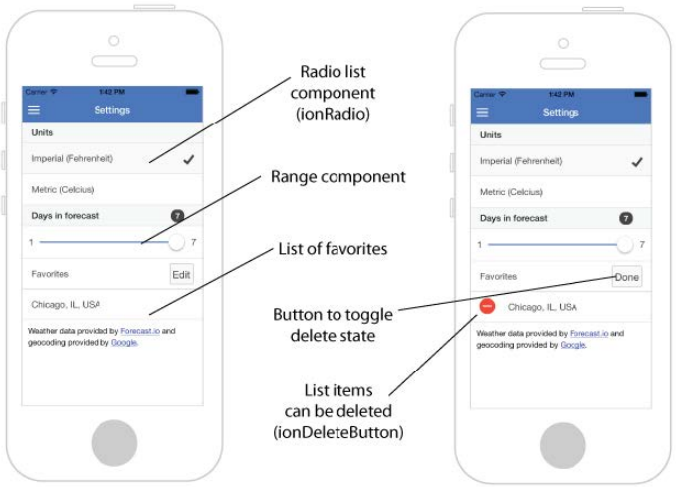


Figure 6.4 – Settings view with a list of radio options, a range input, and a list that can delete items

6.4.1 Create services for locations and settings

Our first step is to create two services, one to keep track of the favorite locations and another for settings. We'll create two services using Angular's factory method that we can then inject into any controller. The Settings service will be just a simple JavaScript object with properties, where the Locations service will contain some methods to help us manage the list of locations.

We will add these two services into the main app JavaScript file in order to keep our example more streamlined, but we could also add these as individual modules. Open up www/js/app.js and add the two new services from listing 6.5 to our app.

Listing 6.5 – Services for Locations and Settings (www/js/app.js)

```
.factory('Settings', function () { #1
  var Settings = { #2
    units: 'us', #2
    days: 8 #2
  }; #2
  return Settings; #2
})

.factory('Locations', function () { #3
  var Locations = { #4
    data: [{ #4
      city: 'Chicago, IL, USA', #4
      lat: 41.8781136, #4
      lng: -87.6297982 #4
    }], #4
    getIndex: function (item) { #5
      var index = -1; #6
      angular.forEach(Locations.data, function (location, i) { #5
        if (item.lat == location.lat && item.lng == location.lng) { #5
          index = i; #5
```

```
        } #5
      }); #5
      return index; #5
    }, #5
    toggle: function (item) { #6
      var index = Locations.getIndex(item); #6
      if (index >= 0) { #6
        Locations.data.splice(index, 1); #6
      } else { #6
        Locations.data.push(item); #6
      } #6
    }, #6
    primary: function (item) { #7
      var index = Locations.getIndex(item); #7
      if (index >= 0) { #7
        Locations.data.splice(index, 1); #7
        Locations.data.splice(0, 0, item); #7
      } else { #7
        Locations.data.unshift(item); #7
      } #7
    } #7
  };

  return Locations; #8
});
```

#1 Declare the Settings service as a factory
#2 Create and return a JavaScript object with default settings
#3 Declare the Locations service as a factory
#4 Create the Locations object, and store a default value for Chicago in data array
#5 Method to determine the index value of a location
#6 Toggle method to add or remove an item from Locations
#7 Primary method will move the item to the top position, or add it to the top if new
#8 Return the Locations object with data and methods

We're using Angular services here to help us define a service that can be shared between different controllers. Later we will be adding each of these to different views, but any changes made to these services will immediately reflect in other views. If you recall from chapter 5, we used the same technique for the list of currencies, where each currency could be toggled on or off and the changes would be reflected instantly across the app. We'll use Locations.data as the array that stores the list of locations, which should contain city name, latitude, and longitude values. To start, I've preset Chicago in the list since it is one of my personal favorite cities.

The Locations service has three methods. The getIndex method gives us the index value of an item from the Locations.data array, if the item exists. The toggle method will add or remove a location from the Locations.data array, after it checks if the location is already in the Locations.data array or not. The last primary method is used to either add a new item to the top of the list, or to move an existing item to the top of the list.

6.4.2 Show favorites in side menu list

Now that we have our Locations service, we can display the list of favorite locations in the side menu. To do this, we need to add a controller for our side menu so we can inject the Locations

into the scope, and then add a new item to the navigation list with ngRepeat to display all of the favorite locations.

First we will define the controller in our app JavaScript file where we just added our services. Since this controller belongs in the side menu instead of an isolated view, I've decided to keep it together with the rest of the main app code. This very simple controller is found in listing 6.6.

Listing 6.6 – Side Menu Controller (www/js/app.js)

```
.controller('LeftMenuCtrl', function ($scope, Locations) { #1
  $scope.locations = Locations.data; #2
})
```

#1 Create a controller and inject services
#2 Assign the locations data array to the scope

This very simple controller just assigns the array of locations to the scope. We don't need to do anything more complex in the controller, but we do need to add this controller to the side menu template. In www/index.html, update the ionSideMenu and add the ngController directive to attach this new controller to the side menu.

```
<ion-side-menu side="left" ng-controller="LeftMenuCtrl">
```

Normally we have not had to use ngController, because we attach the controllers in our state definitions. In this case, the side menu is not a state, so we have to attach the controller ourselves. This means the side menu scope now will have access to the list of locations, so we can now update the list to loop over the favorites. Keep www/index.html open and add the two bolded lines to the list from listing 6.7.

Listing 6.7 – Adding location items to navigation list (www/index.html)

```
<ion-list>
  <ion-item class="item-icon-left" ui-sref="search" menu-close><span class="icon ion-
      search"></span> Find a City</ion-item>
  <ion-item class="item-icon-left" ui-sref="settings" menu-close><span class="icon
      ion-ios7-cog"></span> Settings</ion-item>
  <ion-item class="item-divider">Favorites</ion-item> #1
  <ion-item class="item-icon-left" ui-sref="weather({city: location.city, lat:
      location.lat, lng: location.lng})" menu-close ng-repeat="location in
      locations"><span class="icon ion-ios7-location"></span> {{location.city}}</ion-
      item> #2
</ion-list>
```

#1 Add a divider to display some text
#2 Loop over list of locations, link them to weather state, apply menuClose and display city name

The ngRepeat now loops over the array of locations, and will link to the weather state (which we will define later). Now when you open the side menu, the default Chicago location should appear under Favorites. Later when users are able to add more locations, they will also appear here. Now let's build our settings view.

6.4.3 Adding the settings template

Our settings template will contain three primary areas, a radio list to choose between imperial or metric units, a range input to configure the number of days to show in the forecast, and a list of the favorite locations with the ability to delete items. Let's take a look at the complete code in listing 6.8 and review the several components individually. Create a new file at www/views/settings/settings.html.

Listing 6.8 – Settings template (www/views/settings/settings.html)

```
<ion-view title="Settings">
  <ion-content>
    <ion-list>
      <ion-item class="item-divider">Units</ion-item>
      <ion-radio ng-model="settings.units" ng-value="'us'">Imperial (Fehrenheit)</ion-
      radio> #1
      <ion-radio ng-model="settings.units" ng-value="'si'">Metric (Celcius)</ion-
      radio> #1
      <div class="item item-divider">Days in forecast <span class="badge badge-
      dark">{{settings.days - 1}}</span></div> #2
      <div class="item range range-positive"> #2
        1 <input type="range" name="days" ng-model="settings.days" min="2" max="8"
      value="8"> 7 #2
      </div> #2
      <div class="item item-button-right">Favorites <button class="button button-
      small" ng-click="canDelete = !canDelete">{{(canDelete) ? 'Done' :
      'Edit'}}</button></div> #3
    </ion-list>
    <ion-list show-delete="canDelete"> #4
      <ion-item ng-repeat="location in locations"> #5
        <ion-delete-button class="ion-minus-circled" ng-click="remove($index)"></ion-
      delete-button> #6
        {{location.city}}
      </ion-item>
    </ion-list>
    <p class="padding">Weather data powered by <a
      href="https://developer.forecast.io/docs/v2">Forecast.io</a> and geocoding
      powered by <a
      href="https://developers.google.com/maps/documentation/geocoding/">Google</a>.<
      /p> #7
  </ion-content>
</ion-view>
```

#1 Use ionRadio component to toggle between unit types
#2 Use the input range to set number of days to display
#3 Create a divider with a button that toggles canDelete state
#4 Create a list and show delete buttons based on value of canDelete
#5 Loop over list of locations
#6 Delete button which displays only when delete state is active on list
#7 Credits for the API sources

Let's start with the radio options. The ionRadio component is a wrapped up radio button designed for mobile. Instead of displaying the small circle like it would normally do on a web page, it is restyled as a list with a checkmark to indicate the selected item. It also assumes it is used inside of a list component, so it adopts the same display as a list item. That is why we

don't have to place it inside of a list item. We assign the same ngModel value to both ionRadio inputs, and when the user selects one the other will disable, as you would expect from a radio list.

The next component is an input range slider. This is a newer HTML element to which Ionic applies styling. You see it in figure 6.4 as the line with a circle, which can be drug back and forth to set a value. In this case the options are from values 2-8, but we are displaying the values 1-7. The reason is in our forecast data, we will always show the first day so the setting here is determining how many additional days to display. As you drag the range, the value automatically updates.

Lastly, the last component is the ionList with the option to delete items from the list. The visual experience of showing the delete button is built into the ionList component, but the actual logic to handle the deletion of an item is left up to the developer to implement. To use the delete feature, we use the show-delete="canDelete" attribute. When the expression is true, the delete buttons will appear, otherwise it will hide. You also have to declare an ionDeleteButton inside of each item, and give it a class for the icon you want to use. Here I also use ngClick to call a method on the controller which will take care of removing the item. I've used a button in the item divider that toggles the canDelete value from true to false. The button also uses a more complex expression, which is a ternary operator, changes the text from Edit to Done depending on the value of canDelete.

At the end, we credit the two sources of our data. Sometimes APIs allow you to use their services for free, but ask for credit. To comply with that term, we provide the credit here.

6.4.4 Settings view controller

To finish up our settings view, we need the controller. We will need to access the Locations and Settings services we created earlier, and add the logic to remove a location when the delete button is pressed.

Create a new file at www/views/settings/settings.js and add the controller found in listing 6.9.

Listing 6.9 – Settings controller

```
angular.module('App')
.controller('SettingsCtrl', function ($scope, Settings, Locations) { #1
  $scope.settings = Settings; #2
  $scope.locations = Locations.data; #2
  $scope.canDelete = false; #3

  $scope.remove = function (index) { #4
    Locations.toggle(Locations.data[index]); #4
  }; #4
});
```

#1 Declare the controller and inject services
#2 Set the settings and locations data on the scope
#3 Set the default state for if the list can delete or not
#4 Method to handle removing an item from the list of locations

This controller is fairly simple, since we essentially only do two things. First we set some values on the scope, which some come from the services we defined. Remember, these are JavaScript objects we created earlier and any changes made in the settings view to those values will be reflected elsewhere. The remove method takes the index value of the location, and then calls the Locations.toggle method with the item to remove. Since we abstracted the adding and removing of locations into the Locations service, we don't have to rewrite the logic here.

The last two things we need to do now are to add a new state for settings, and make sure we add the settings controller to our application. Start by opening up www/index.html and add a new script tag for our controller after the other script tags.

```
<script src="views/settings/settings.js"></script>
```

Then open up www/js/app.js and declare the state for settings as you see in listing 6.10. This is the final step before we can see the settings view in action. Add this into the state provider declaration.

Listing 6.10 – Settings view state declaration (www/js/app.js)

```
.state('settings', {
  url: '/settings',
  controller: 'SettingsCtrl',
  templateUrl: 'views/settings/settings.html'
})
```

We've finished the settings view, which contained two Ionic form components, radio items and a range input, as well as a list with the ability to delete items.

6.5 Setup weather view

The last view we will setup is the weather view, which is designed to display the current weather and forecast for a location. In this section we'll create the base for the weather view, and then add more complexity to it in the remaining sections of this chapter. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step5
```

The result of this section can be seen in figure 6.5. The view will be fairly simple at this point, but we'll be adding more design and content as we go.

Figure 6.5 – Weather view base loading from Forecast.io and showing current temperature

6.5.1 Get a Forecast.io API key

The Forecast.io service requires an API key to make requests. It only requires you to create an account with an email address and password, and, unless you want to use their paid service, does not require a credit card or other personal information. Go to https://developer.forecast.io/ and sign up for your free account to get your token. You will need this token in a few moments.

6.5.2 Using Ionic CLI proxies

Forecast.io does not support CORS as of this writing, which means that by default you cannot load data from their API in a browser. CORS (Cross-Origin Resource Sharing) is a set of security rules that browsers implement so that your web applications can load data from another domain. By default, browsers will block access to loading data from another domain since you cannot trust what that external domain will send you. However, if you trust the source of data and the API supports CORS, you are able to load that data. You can read more about CORS at http://enable-cors.org or get the book CORS in Action at http://manning.com/hossain. The other RESTful APIs that we've used in this book have all supported CORS and therefore haven't required any additional work on our end.

The Ionic CLI utility provides a feature that allows you to bypass this limitation by using a proxy. Essentially it allows you to create something like a shortcut or alias URL that is attached to the server that the `ionic serve` command sets up and can pass your original request through to the real API URL. In a production application, you will still need to properly address the CORS limitations by the API in another way.

In your app, there is an ionic.project file that you should open. This file contains a JSON object for configuring the Ionic project, and here we can define a new property to map a list of

URLs to proxy. Keeping the JSON valid, add the bolded part to your ionic.project file. It may contain other information than is shown here in listing 6.11.

Listing 6.11 – Declare proxy in ionic.project file (ionic.project)

```
{
  "name": "chapter6",
  "app_id": "",
  "proxies": [ #1
    {
      "path": "/api/forecast", #2
      "proxyUrl": "https://api.forecast.io/forecast/e33b593dcdddc5bfaa38a4babc79d95a/"
       #3
    }
  ]
}
```

#1 Add a new proxies property, which is an array of objects
#2 Add a path, which will be the new proxy URL to call in your app
#3 Add a proxyUrl property, which is the endpoint that will be called

We've just declared a proxy, so in our app we can call /api/forecast and it will actually proxy through our local server and go to the proxyUrl defined above. The part of the proxyUrl that is a long string of random numbers and letters is where you will need to put your API key from Forecast.io. This is a key I used during writing, but you should use your own.

The next time the ionic serve command is run, the proxy will be setup. It will also work if you are using ionic emulate or ionic run with the live reload option turned on. This will allow us to develop locally with the Forecast.io service, and you can use the same technique for building apps locally with other services that don't support CORS.

6.5.3 Setup the weather view

We will now get the weather view added to our application. We'll just get the forecast to load and display the current temperature. Later we'll be adding a number of Ionic components and content to this view.

First, let's tackle the template. This is very vanilla, it will display the name of the location in the header bar and show the current temperature. In figure 6.5 you can see it is a chilly 18 degrees Fahrenheit in the Windy City. Create a new file at www/views/weather.weather.html and use the code from listing 6.12.

Listing 6.12 – Base weather view template (www/views/weather/weather.html)

```
<ion-view title="{{params.city}}">
  <ion-content>
    <h3>Current Conditions</h3>
    <p>{{forecast.currently.temperature | number:0}}&deg;</p>
  </ion-content>
</ion-view>
```

There isn't too much going on here, we're just binding some data into the title and the content areas. That data will be loaded in the controller next. We are using the number filter on

the temperature to round the value to a whole number, since the value returned by the service is more accurate for our needs. We are assuming that users will expect the temperature value to be given in whole numbers.

Now let's add the controller. It also is fairly light at the moment, but we will expand it slowly as we go along. Listing 6.13 contains the controller that you will need to add into a new file at www/views/weather/weather.js.

Listing 6.13 – Weather controller (www/views/weather/weather.js)

```
angular.module('App')
.controller('WeatherCtrl', function ($scope, $http, $stateParams, Settings) { #1
  $scope.params = $stateParams; #2
  $scope.settings = Settings; #2

  $http.get('/api/forecast/' + $stateParams.lat + ',' + $stateParams.lng, {params:
      {units: Settings.units}}).success(function (forecast) { #3
    $scope.forecast = forecast; #3
  }); #3
});
```

#1 Define the controller and inject services
#2 Attach service data to the scope
#3 Make the HTTP request to load forecast

When this controller executes, it will first store some values on the scope. The $stateParams are assigned to the scope and used to get the location name for the header bar. Then the settings are also set on the scope, for future use. Lastly, it makes an HTTP request to our proxy URL, adding the latitude, longitude, and units type for the request. When it returns, it stores the forecast on the scope for the template to use.

Lastly, we need to add the new state to our state provider list, and add the controller to the index.html. Open the index.html and add the script tag below to include the controller after the other script tags.

```
<script src="views/weather/weather.js"></script>
```

Then open up www/js/app.js and add the last state for weather from listing 6.14. This will finish the

Listing 6.14 – Weather view state declaration (www/js/app.js)

```
.state('weather', {
  url: '/weather/:city/:lat/:lng',
  controller: 'WeatherCtrl',
  templateUrl: 'views/weather/weather.html'
});
```

In the next section we will build out a paginated scrolling view for our forecast information.

6.6 ionScroll: Building custom scrolling content

This section is focused on giving a custom scrolling experience for our forecast data, and adding the necessary markup and styling to give it a nice appearance. Since there are so many weather apps available, it is important to craft a good user experience. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step6
```

We are going to use ionScroll to create a pagination vertical scroller. This means as the user swipes up or down, the scrolling will always continue until the next page. In some ways this is like the ionSlideBox, but vertically and with a slightly different experience. Then we'll add the content and styling for each of the pages in the scroller. Lastly, we'll add a few filters to help format our data in a more meaningful way.

At the end of the chapter, you'll be able to scroll through a weather forecast like you see in figure 6.6. The scrolling only stops when it reaches the next page.

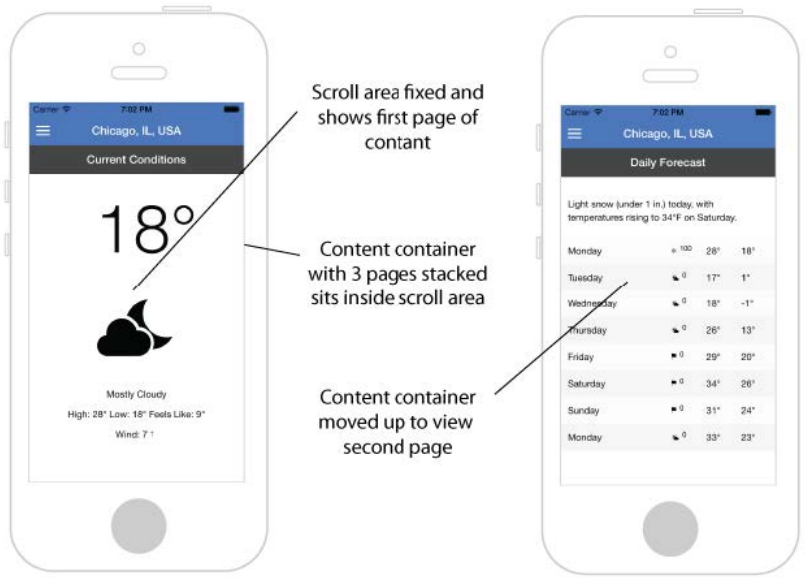


Figure 6.6 – Using ionScroll with paging enabled for separated forecast content

6.6.1 Using ionScroll with paging

First we need to create the scrolling experience with ionScroll. Normally we have just used ionContent and our content has always filled the space and allowed scrolling. However, ionScroll gives us a little more control over how the scrolling content area functions, and in this case it provides the paging feature we desire. ionScroll is most useful in cases where you want

to have more control over the scrolling experience, manage how the user can zoom, or lock scrollable direction.

The ionScroll directive has to be given a width and height value. This is something ionContent does for you automatically, but ionScroll does not. Since our apps can be loaded onto different devices with different screen sizes, we have to be able to calculate the size of ionScroll based on the size of the screen.

For our scrolling area, we will make the ionScroll the same size as the viewable area, and then create a div element inside of ionScroll with 3x the height of the ionScroll. This div inside will be able to slide up or down, to give us the scrolling effect we are after. We will lock scrolling to vertically only, and with the paging feature enabled it will also scroll until it hits the next page. Imagine the ionScroll was 500px tall, the div inside would be 1500px tall and have 3 pages (500 * 3 = 1500). Since it is 500px tall, when paging is enabled scrolling will always stop on a boundary that is based on the ionScroll height, in this case 0px (page 1), 500px (page 2), or 1000px (page 3). Look at figure 6.7 to see how these layers work to create the scrolling experience.

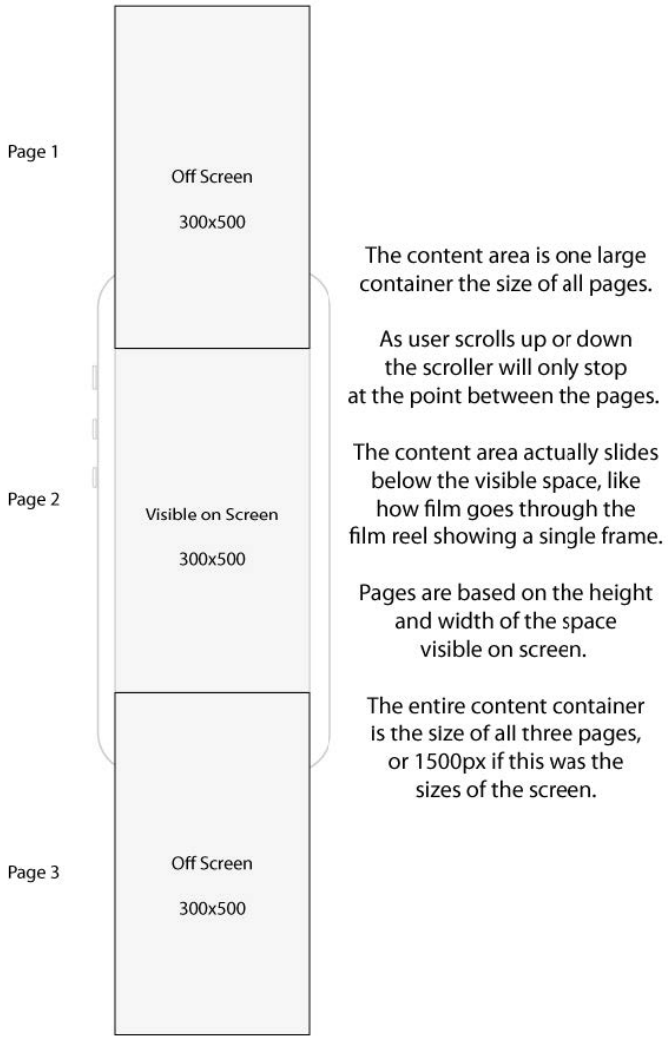


Figure 6.7 – How ionScroll with paging will allow us to scroll by page

Let's start by looking at the template with ionScroll included. It uses some calculations that will be added to the controller soon, so it will not function until we've updated the controller.

Listing 6.15 – Weather template with ionScroll (www/views/weather/weather.html)

```
<ion-view title="{{params.city}}">
  <ion-content> #1
    <ion-scroll direction="y" paging="true" ng-style="{width: getWidth(), height:
      getHeight()}"> #2
```

```
      <div ng-style="{height: getTotalHeight()}"> #3
        <div class="scroll-page page1" ng-style="{width: getWidth(), height:
      getHeight()}"> #4
          Page 1 #4
        </div> #4
        <div class="scroll-page page2" ng-style="{width: getWidth(), height:
      getHeight()}"> #4
          Page 2 #4
        </div> #4
        <div class="scroll-page page3" ng-style="{width: getWidth(), height:
      getHeight()}"> #4
          Page 3 #4
        </div> #4
      </div>
    </ion-scroll>
  </ion-content>
</ion-view>
```

#1 Use ionContent here to position ionScroll properly
#2 Use ionScroll and lock direction to vertically only with paging, give exact height and width styling
#3 Create the inner div and give it a height equal to all 3 page stacked
#4 Pages each declared with the same width and height of the ionScroll area

You'll see we use the ionContent and then put the ionScroll inside of it. The ionContent will give us a container that takes into account the size of the header bar. Inside of that the ionScroll exists, and it gets the height and width from the controller calculations. The ionContent will not actually scroll, because the ionScroll will be the exact size of the visible space.

Then, we have one div inside of ionScroll, and it has the total height of all three of the scroll pages. The scroll pages are then inside, and are stacked on top of one another, like you see in figure 6.7.

Now let's add the controller methods used to calculate sizes, so we can preview how it scrolls. Open the controller at www/views/weather/weather.js and add the following lines from listing 6.16 inside the controller.

Listing 6.16 – Controller methods to determine sizes (www/views/weather/weather.js)

```
var barHeight = document.getElementsByTagName('ion-header-bar')[0].clientHeight; #1
$scope.getWidth = function () { #2
  return window.innerWidth + 'px'; #2
}; #2
$scope.getTotalHeight = function () { #3
  return parseInt(parseInt($scope.getHeight()) * 3) + 'px'; #3
}; #3
$scope.getHeight = function () { #4
  return parseInt(window.innerHeight - barHeight) + 'px'; #4
}; #4
```

#1 Get the first header bar's height
#2 Give items the width of the app
#3 Get the total height by multiplying the height of the space by the number of pages
#4 Give items the height of the app without the header bar

We start off by getting the height of the header bar, since this may vary from platform to platform. The getWidth, getHeight, and getTotalHeight methods then use the size of the window itself to determine the amount of space available, minus the bar height. This programmatic approach to determining the size is required only because different devices have different screen sizes, and we want our pages to be the same size as the screen. You can create a scrolling region with items of a fixed size to scroll through them using the same logic but providing an explicit size.

Now that we understand the scrolling, let's add the content for each page into the scrolling pages. Listing 6.17 has the updated template for www/views/weather/weather.html.

Listing 6.17 – Content for weather template (www/views/weather/weather.html)

```
<ion-view title="{{params.city}}">
  <ion-content>
    <ion-scroll direction="y" paging="true" ng-style="{width: getWidth(), height:
      getHeight()}">
      <div ng-style="{height: getTotalHeight()}">
        <div class="scroll-page center" ng-style="{width: getWidth(), height:
      getHeight()}">
          <div class="bar bar-dark"> #1
            <h1 class="title">Current Conditions</h1> #1
          </div> #1
          <div class="has-header"> #2
            <h2 class="primary">{{forecast.currently.temperature |
      number:0}}&deg;</h2>
            <h2 class="secondary icon" ng-class="forecast.currently.icon |
      icons"></h2> #3
            <p>{{forecast.currently.summary}}</p>
            <p>High: {{forecast.daily.data[0].temperatureMax | number:0}}&deg; Low:
      {{forecast.daily.data[0].temperatureMin | number:0}}&deg; Feels Like:
      {{forecast.currently.apparentTemperature | number:0}}&deg;</p>
            <p>Wind: {{forecast.currently.windSpeed | number:0}} <span class="icon
      wind-icon ion-ios7-arrow-thin-up" ng-style="{transform: 'rotate(' +
      forecast.currently.windBearing + 'deg)'}"></span></p> #4
          </div>
        </div>
        <div class="scroll-page" ng-style="{width: getWidth(), height: getHeight()}">
          <div class="bar bar-dark">
            <h1 class="title">Daily Forecast</h1>
          </div>
          <div class="has-header">
            <p class="padding">{{forecast.daily.summary}}</p>
            <div class="row" ng-repeat="day in forecast.daily.data |
      limitTo:settings.days"> #5
              <div class="col col-50">{{day.time + '000' | date:'EEEE'}}</div> #6
              <div class="col"><span class="icon" ng-class="day.icon |
      icons"></span><sup>{{day.precipProbability | chance}}</sup></div> #7
              <div class="col">{{day.temperatureMax | number:0}}&deg;</div>
              <div class="col">{{day.temperatureMin | number:0}}&deg;</div>
            </div>
          </div>
        </div>
        <div class="scroll-page" ng-style="{width: getWidth(), height: getHeight()}">
          <div class="bar bar-dark">
            <h1 class="title">Weather Stats</h1>
          </div>
          <div class="list has-header">
```

```
            <div class="item">Sunrise: {{forecast.daily.data[0].sunriseTime |
        timezone:forecast.timezone}}</div> #8
            <div class="item">Sunset: {{forecast.daily.data[0].sunsetTime |
        timezone:forecast.timezone}}</div> #8
            <div class="item">Visibility: {{forecast.currently.visibility}}</div>
            <div class="item">Humidity: {{forecast.currently.humidity * 100}}%</div>
          </div>
        </div>
      </div>
    </ion-scroll>
  </ion-content>
</ion-view>
```

#1 Using a header bar like a subheader
#2 Use the has-header class to position the content of this page
#3 Use the icons filter to map to an Ionicon based on conditions
#4 Using the wind direction given in degrees to rotate an arrow to point that direction
#5 Using the limitTo filter to only show the number of days from settings
#6 Using the date filter to convert a unix timestamp into the day of the week
#7 Using the chance filter to round the percentage to a 10 value
#8 Getting the sunrise/sunset time, which is converted into location timezone

The template has a lot of content, but it is mostly binding data into the view and elements used for styling and positioning. Each page has a bar element, which contains the title for that page. Then inside of the following element we have different content for each. Until we create the filters, the application will not be able to correctly load.

Inside of the second page, you'll see a neat Ionic component that we haven't used before. There is a div with the row class, and then several divs inside with the col class. If you are familiar with CSS frameworks like Bootstrap, you'll recognize a CSS grid system being used. In other words, the CSS grid is like an auto adjusting layout with rows and columns. In this case, we have 4 columns, and we've set the first column to take 50% of the width. Ionic's CSS grid component uses the CSS flexbox feature to automatically adjust the layout of columns, so if you don't specify a specific column width the columns are automatically equally sized with any remaining space.

Right now it will look a little disorganized. We need to add some CSS to get the design to look a little smoother. Open up the www/css/styles.css file and add the CSS rules from listing 6.18.

Listing 6.18 – Styling for weather view (www/css/styles.css)

```
.scroll-page .icon:before {
  padding-right: 5px;
}
.scroll-page .row + .row {
  margin-top: 0;
  padding-top: 5px;
}
.scroll-page .row:nth-of-type(odd) {
  background: #fafafa;
}
.scroll-page .row:nth-of-type(even) {
  background: #f3f3f3;
```

```
}
.scroll-page .wind-icon {
  display: inline-block;
}
.scroll-page.center {
  text-align: center;
}
.scroll-page .primary {
  margin: 0;
  font-size: 100px;
  font-weight: lighter;
  padding-left: 30px;
}
.scroll-page .secondary {
  margin: 0;
  font-size: 150px;
  font-weight: lighter;
}
.scroll-page .has-header {
  position: relative;
}
```

These CSS rules apply to the contents of the scrolling pages only, and are just used to give some cleaner display to the elements. In your apps, you will likely write more CSS than we have in our examples, but our focus is always on Ionic's features.

6.6.2 Creating filters for forecast data

The forecast data has a few formatting problems that we'd like to fix or improve. First, we'd like to show an icon related to the weather forecast. For example, if it is raining, we'd like to use one of Ionic's rainy icons. Since we've already declared the filters in our template,

Then we'd like to modify the chance of rain values to always round them to the nearest 10th value. Normally the chance of rain is reported like 20%, not 17%.

Lastly, we need to fix the sunrise and sunset timestamps. They are currently shown based on your timezone. For example, if you live in Chicago and viewed the weather in London you would see the local version of the time from Chicago. This is confusing, because in that case the sunrise would be in the middle of the night. We'll use a JavaScript library called Moment to help us manage the timezone and display the times according to the location's timezone and not the user's timezone.

First, we'll install the Moment library files using ionic add. You can install the files quickly using the command below from the root of your project.

```
$ ionic add moment-timezone
```

It will take a moment to download and install Moment and MomentTimezone, which we need to use both to correctly manage timezones. When they are complete, add them to your index.html after the ionic script tag and before the script tags for your app files.

```
<script src="lib/moment/moment.js"></script>
<script src="lib/moment-timezone/builds/moment-timezone-with-data.js"></script>
```

Now we have the Moment library setup, we can create a new filter that will handle converting the timestamp into the correct timezone of the location. We are lucky because the forecast data provides us with the timezone of the location, so we don't need to do anything special to find it.

Listing 6.19 has the code for all three filters that we want to create. Open up www/js/app.js and add these three filters as part of your app.

Listing 6.19 – Filters for weather view

```
.filter('timezone', function () { #1
  return function (input, timezone) {
    if (input && timezone) { #2
      var time = moment.tz(input * 1000, timezone); #2
      return time.format('LT'); #2
    } #2
    return ''; #2
  };
})

.filter('chance', function () { #3
  return function (chance) {
    if (chance) { #4
      var value = Math.round(chance * 10); #4
      return value * 10; #4
    } #4
    return 0; #4
  };
})

.filter('icons', function () { #5
  var map = { #6
    'clear-day': 'ion-ios7-sunny', #6
    'clear-night': 'ion-ios7-moon', #6
    rain: 'ion-ios7-rainy', #6
    snow: 'ion-ios7-snowy', #6
    sleet: 'ion-ios7-rainy', #6
    wind: 'ion-ios7-flag', #6
    fog: 'ion-ios7-cloud', #6
    cloudy: 'ion-ios7-cloudy', #6
    'partly-cloudy-day': 'ion-ios7-partlysunny', #6
    'partly-cloudy-night': 'ion-ios7-cloudy-night' #6
  }; #6
  return function (icon) { #6
    return map[icon] || ''; #6
  } #6
})
```

#1 Create the timezone filter, to convert to location timezone
#2 Only if a timestamp and timezone are provided will it convert a timestamp based on a timezone
#3 Create the chance filter, to convert to a percentage chance for precipitation
#4 If a value is given, round the percentage to a multiple of 10
#5 Create the icons filter, to convert a condition type into an ionicon
#6 Based on a map of conditions to ionicons, return the icon if found

You could have put this logic into the controller, but then they are not very easy to reuse elsewhere. The filters here are fairly straightforward. The timezone filter will convert a

timestamp to display based on a specific timezone. The chance filter will take a percentage value and round it to the nearest 10s value. The icons filter will take the icon value from the forecast data, and map it to an Ionicon.

Now our app will be able to run and display the full weather forecast. With the ionScroll component we are able to scroll up and down, but it will always stop at the next page. This section may have been a bit complex, but looking at each individual component should help clarify what it is doing. Our next step is to create a new option button that will open up an action sheet with a list of options for our users.

6.7 Action Sheet: Displaying a list of options

The action sheet component is another useful tool when you wish to display a list of options to a user. In our situation we would like to display a list of options for the user so they can toggle the location as a favorite or they can set the location as the primary location. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step7
```

In order to show a list of options, the action sheet is a list of buttons that slides up from the bottom of the screen. Usually there is a cancel button, and sometimes there is a special button that indicates a destructive action such as deleting. If you tap on the area outside of the buttons it will close the sheet, much like a modal or popover. You can see the action sheet in action in figure 6.8.

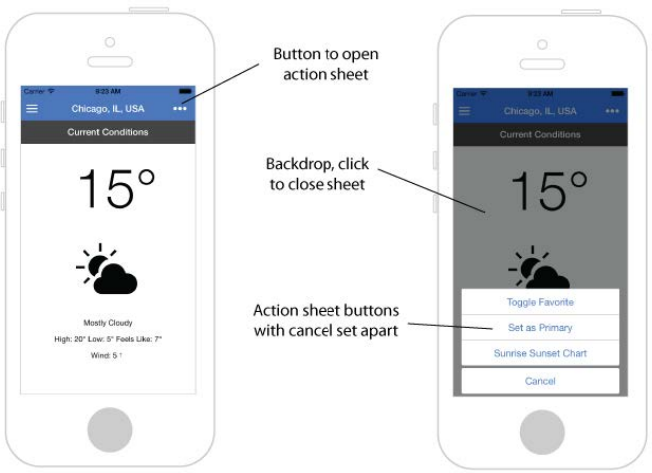


Figure 6.8 – Button that then opens the action sheet with options

The action sheet component is something iOS users will find familiar, but Android does not have a similar native component. You should consider the implications of using this feature

when you plan to also support Android and, while there is no technical reason it won't work on Android, it might not be the most intuitive for Android users.

There is no template for the action sheet, it is entirely run from the $ionicActionSheet service. You'll need to declare the list of buttons, and what to depending on which button was selected. We need to start by adding a button to our view that will be able to trigger the action sheet, found in listing 6.20.

Listing 6.20 – Action sheet more button (www/views/weather/weather.html)

```
<ion-view title="{{params.city}}">
  <ion-nav-buttons side="left"> #1
    <button class="button button-clear" menu-toggle="left"><span class="icon ion-
      navicon"></span></button> #1
  </ion-nav-buttons> #1
  <ion-nav-buttons side="right"> #2
    <button class="button button-icon" ng-click="showOptions()"><span class="icon ion-
      more"></span></button> #3
  </ion-nav-buttons>
  <ion-content> #4
```

#1 Redeclare sidebar toggle left button
#2 Add new nav button to right side
#3 Have a button call the showOptions method when used
#4 Continue with rest of the template

This will add a new button to the right side of the title in the header bar that has the more icon, which is three dots. It will call a method in the controller that we will write next, and that method will handle the logic to open the action sheet. We also added the same left button again, otherwise it would be replaced since we declared a new button for this view.

To do this, you'll need to update the controller in www/views/weather/weather.js. We'll need to also inject the $ionicActionSheet service into the controller, which is not shown here. See the code for this method in listing 6.21.

Listing 6.21 – Action sheet in controller (www/views/weather/weather.js)

```
$scope.showOptions = function () {
  var sheet = $ionicActionSheet.show({ #1
    buttons: [ #2
      {text: 'Toggle Favorite'}, #2
      {text: 'Set as Primary'}, #2
      {text: 'Sunrise Sunset Chart'} #2
    ],
    cancelText: 'Cancel', #3
    buttonClicked: function (index) { #4
      if (index === 0) { #5
        Locations.toggle($stateParams); #5
      } #5
      if (index === 1) { #6
        Locations.primary($stateParams); #6
      } #6
      if (index === 2) { #7
        $scope.showModal(); #7
      } #7
      return true; #8
```

```
    }
  });
};
```

#1 Use the show method to setup and show an action sheet, must have injected $ionicActionSheet
#2 An array of objects for the buttons, object must have a text property
#3 Show the optional Cancel button and give it text
#4 Method to handle button clicks, index of the selected button is provided
#5 This will use the Location service to toggle the current location as favorite
#6 This will use the Locations service to set the current location as primary
#7 We will add something here in the next section to open a modal
#8 Returning true will close the action sheet, otherwise it will remain open

This controller method is called by our button, and action sheet is immediately told to show itself. We are assigning the value of $ionicActionSheet.show() to the sheet variable, which returns a function that can close the sheet. At any point we could call sheet() to close it. The show method takes an object with various properties, and here we've chosen to create three buttons in the buttons array, plus the cancel button which is defined separately. The cancel and destructive (not used here) buttons are separate properties because they are special buttons. By default, cancel will just close the sheet, which is the behavior we have adopted here. You can optionally handle the cancel (and destructive) button click events yourself.

The last property is the buttonClicked function. This is called with a button is selected, and provides the index of the button from the first list. If the cancel or destructive buttons are selected this function does not execute because they have their own versions, not shown here. Since we have three buttons, we have three conditionals to check the value of index and execute logic based on the button. The first two buttons use the Locations service we created earlier, but the third will not do anything just yet. We'll be adding that button in the next section.

That is all you need to have the action sheet component in your app. Now to address that last button, we want it to open up a modal.

6.8 ionModal: Displaying the sunrise and sunset chart

Modals are used heavily in user interfaces today. Modals are a temporary view that is layered above the current view. On websites, they are often used to prompt a user to sign up for a newsletter or to force focus on a subset of content that darkens the rest of the content. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step8
```

In mobile, a modal may be used in slightly different contexts but the principles remain. The main draw of the modal is the fact that it can open above the current content, but be closed to return. Some example modal uses include showing a preview of a search result item without leaving the search result page, opening a modal with a list of additional search filters, or to display an alert or notice for a weather event. You can see the modal in action in figure 6.9.

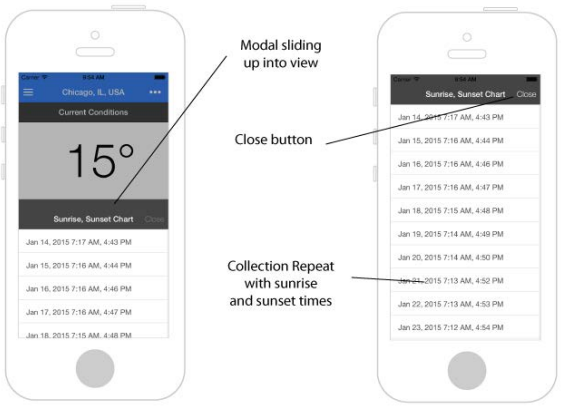


Figure 6.9 – The modal opens by sliding up from the bottom and overlaying the entire app

Modals are designed to overlay the entire app when the app is running on a smaller phone device. If the app runs on a larger tablet device, the modal will not actually fill the entire app but will float in the middle. You can modify the exact size with CSS, but by default the size of the modal is a percentage of the size of the screen on tablets. This is important because it means without some custom styling the modal window size will vary from device to device.

6.8.1 Setting up a modal

In our example, the modal is going to show the sunrise and sunset chart for the location. We'll start by using the $ionicModal service to create a new modal instance, and then update the third button in the action sheet to trigger opening the modal. Much like the popover, we also have to clean up the modal when the scope is destroyed to prevent memory leaks.

To begin, open up the controller in www/views/weather/weather.js and inject the $ionicModal service to the controller, which is not shown in listing 6.22.

Listing 6.22 – Modal in weather view (www/views/weather/weather.js)

```
$ionicModal.fromTemplateUrl('views/weather/modal-chart.html', { #1
    scope: $scope #1
  }).then(function (modal) { #2
    $scope.modal = modal; #2
  }); #2
  $scope.showModal = function () { #3
    if ($scope.modal) { #3
      $scope.modal.show(); #3
    } #3
  }; #3
  $scope.hideModal = function () { #4
    $scope.modal.hide(); #4
  }; #4
  $scope.$on('$destroy', function() { #5
    $scope.modal.remove(); #5
  }); #5
```

#1 Create a new modal from a template url, pass current scope to it, must inject the $ionicModal service first
#2 When template is loaded, assign the modal to the scope to use it
#3 Method to show the modal, only after it has loaded properly
#4 Method to hide the modal
#5 When the current view is destroyed, also remove the modal from memory

This syntax is nearly identical to the ionPopover syntax we saw in chapter 5. Modals are isolated views, which means they need a new template. In this case, we are loading that template from a URL, but you can also provide that template inline. I recommend using this method to avoid writing HTML inside of JavaScript.

The fromTemplateUrl method takes two arguments, a string for the template URL and an object. You can specify additional options, such as the type of animation or if the hardware back button can close the modal (for some Android devices). Modals create an isolated child scope, and the scope parameter tells the modal which scope should be the parent for the modal. By default it is the root scope, and we want this modal to have access to our weather scope so we assign it as the parent instead.

Since loading a template is asynchronous, it returns a promise which we resolve with then. It gives us an instance of a modal controller object, which has properties like show, hide to control the modal. I have the showModal method check first if the modal exists before trying to open it, to prevent the situation where the modal hasn't completed loading yet and throwing an error.

The scope destroy event is something you must do with each modal you create, or else it will always remain in memory. Most of the Ionic components are able to clean up after themselves, but due to the way the modal is designed this is not possible.

To get the modal to actually appear, we need to create the template file for it. Create a new file at www/views/weather/modal-chart.html and add the contents from listing 6.23.

Listing 6.23 – Modal contents template (www/views/weather/modal-chart.html)

```
<ion-modal-view> #1
  <ion-header-bar class="bar-dark"> #2
    <h1 class="title">Sunrise, Sunset Chart</h1> #2
    <button class="button button-clear" ng-click="hideModal()">Close</button> #2
  </ion-header-bar> #2
  <ion-content> #3
  </ion-content> #3
</ion-modal-view>
```

#1 ionModalView must wrap the contents of the modal template
#2 Add a header bar with a close button
#3 Empty content area, for the moment

Using ionModalView is a specialized version of ionView that we use elsewhere, and is required when making a modal template. Just make sure to wrap your modal template in ionModalView to properly get the design and positioning for a modal window.

Since it is a blank view, we add a header bar and content area. The header bar has a close button that calls the hideModal method, which is from the parent scope (the weather scope). We want to have some content, so let's add the content for the sunrise and sunsets.

6.8.2 Collection Repeat: Making sunrise and sunset list fast

We want to show the sunrise and sunset for the entire year, which we can calculate using the aid of a helpful library called SunCalc. Since the sunrise and sunset cycle repeats yearly, we only need to show the chart for one year.

We can use a normal list with ngRepeat to create the long list of items, but that means we would be creating 365 items when only a small number can actually appear on the screen at once. If we create all 365 items in a list, they all still render and take up memory regardless if they are on or off screen. This will impact performance of our list, mostly from having to render too many DOM elements, and can cause it to be slow to display or have poor scrolling smoothness.

To address this, we will be using the collection repeat feature. Instead of creating 365 items, it will create just enough to display on the screen. When the user scrolls, it will destroy items that go out of view and create new items and add them to the list. This is going to give us much better memory management, and most importantly provide a smoother scrolling experience. Any large set of data that you want to scroll through can benefit from using collection repeat.

There are some caveats to using collection repeat. It only works with arrays of items, which means you can't have an object and use collection repeat. Also it requires you to define the exact height and width of each item in the list when displayed. In order to properly calculate the necessary number of items to display, each item must declare its own size. Items do not have to be the same size however. The collection repeat will take up the entire size of its container. Lastly, to make the styling work well, you might need to be careful about what CSS you use in your items and avoid doing anything that will show or hide items in the list.

Before we can use the collection repeat, we need to create an array with our year of sunrise and sunset times. We need to install the SunCalc library.

```
$ ionic add suncalc
```

Then we can include the library in our index.html file by adding the script tag after the other library scripts.

```
<script src="lib/suncalc/suncalc.js"></script>
```

Lastly, we will create the chart when the modal is requested. We will update the showModal method in the weather controller to generate the list of times for the next year before the modal appears. Open www/views/weather/weather.js and update the showModal method as in listing 6.24.

Listing 6.24 – Generating chart (www/views/weather/weather.js)

```
$scope.showModal = function () {
  if ($scope.modal) {
    var days = [];
    var day = Date.now();
    for (var i = 0; i < 365; i++) { #1
      day += 1000 * 60 * 60 * 24; #1
      days.push(SunCalc.getTimes(day, $scope.params.lat, $scope.params.lng)); #2
    }
    $scope.chart = days; #3
    $scope.modal.show();
  }
};
```

#1 For each day, add another day to the timestamp
#2 Use SunCalc to get the times based on latitude, longitude and day
#3 Assign the list of days to the scope

Here we are creating an array with the times for each day of the year, starting from tomorrow for a whole year. SunCalc requires a timestamp, latitude, and longitude value to be able to calculate the sunrise and sunset values. Those values are pushed into the array, and then stored on the scope since the modal will need to access the chart array for the collection repeat.

Now to implement the collection repeat, we need to open our modal template again. Edit the www/views/weather/modal-chart.html file and update the content with the contents of listing 6.25.

Listing 6.25 – Collection repeat in action (www/views/weather/modal-chart.html)

```
<ion-content>
  <div class="list"> #1
    <div class="item" style="width: 100%" collection-repeat="day in chart" collection-
      item-width="getWidth()" collection-item-height="'53px'"> #2
      {{day.sunrise | date:'MMM d'}}: {{day.sunrise | date:'shortTime'}}, {{day.sunset
      | date:'shortTime'}} #3
    </div>
  </div>
</ion-content>
```

#1 Using a list to contain the items
#2 Use collection repeat, like ngRepeat and define the width and height for each item
#3 Bind the date, sunrise, and sunset times to the list item

Collection repeat is implemented with the same syntax as ngRepeat, "item in array", though it does support some other more complex expressions listed in the documentation. We've used the list component to style our items, though this doesn't matter to the collection repeat. It only needs to know the size, and I use the same getWidth method from earlier to get the width of the window, and give each item a fixed height of 53px. Then we bind the data into the view using the date filters.

As long as you are able to provide the exact size of the collection items, collection repeat has far better performance on large data sets. In this case, our set became large enough to see some lag and the collection repeat directive provides a much better experience.

We'll add one last feature, a popup to confirm or alert the user when they change their favorite locations.

6.9 Popup: Alert and confirm changes to favorites

Right now when you select the Toggle Favorites button in the action sheet, it silently updates the choice for you without telling you what happened. Users appreciate getting visual feedback about their changes, and one way to do this is to use a popup. If you are following along using Git, you can check out the code for this step.

```
$ git checkout -f step9
```

Popups are familiar to web users, because they appear with a message and a button or two asking you questions like "Are you sure?" or alerting you saying "We are sorry an error occurred.". Ionic provides three types of popup defaults or the choice to design your own. The three types are alert, confirm, and prompt. Alerts are meant to simply convey information, such as a message about success or failure to complete an action. Confirms are meant to verify the user meant to do something, such as confirm that you meant to delete the item. Prompts are designed to ask the user for some information, such as a title for an item you are about to save. You can see the alert and confirm options in figure 6.10.

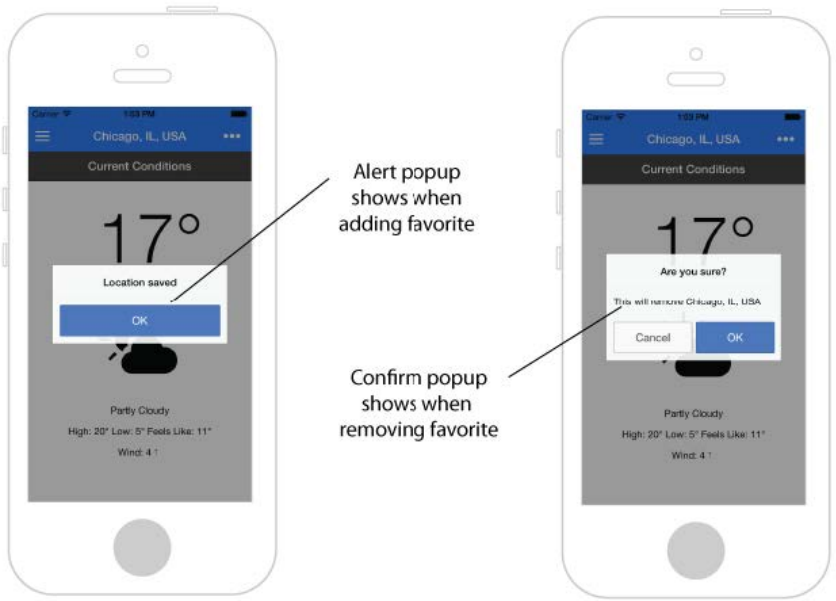


Figure 6.10 – Popups with the left an alert and the right is confirm

Popups should be used with caution, because they are interruptive to the user experience. There are many ways to design an interface to provide feedback to the user, and popups are best used when you want to be completely sure the user has read the message or when you need to prompt for feedback before continuing.

To add the popup, we are going to add it into the Locations service toggle method. We will confirm the user intends to remove the favorite, and alert them when the favorite was added. Right now when the user toggles a favorite location, it is done in the background and the user does not get any feedback or confirmation it was completed.

Open up the ww/js/app.js file and locate the Locations service. First we need to inject the $ionicPopup service into the Locations service.

Now replace the existing toggle method with the code from listing 6.25, which will add both an alert and confirm popup during toggle.

Listing 6.25 – Using ionPopup to alert and confirm (www/js/app.js)

```
toggle: function (item) {
  var index = Locations.getIndex(item);
  if (index >= 0) {
    $ionicPopup.confirm({ #1
      title: 'Are you sure?', #2
      template: 'This will remove ' + Locations.data[index].city #2
    }).then(function (res) { #3
      if (res) { #3
        Locations.data.splice(index, 1); #3
      } #3
    }); #3
  } else {
    Locations.data.push(item);
    $ionicPopup.alert({ #4
      title: 'Location saved' #4
    }); #4
  }
},
```

#1 Create a confirm popup and pass object to define, it will by default have ok and cancel buttons
#2 Give the title and content of the popup
#3 When a button is selected, the function is called and res will be true when ok was selected to delete the item
#4 Create an alert popup with a title, it will by default have just the ok button

The confirm popup is used to verify that the user wishes to delete the item, and unless you override the settings in the configuration object it will have two buttons, OK and Cancel. When a button is selected, the promise will resolve the function and pass the value of res as Boolean. If the user selects OK, res will be true and we then proceed to handle deleting the item.

The alert popup is fired after the item was already added, and just informs the user the location was saved. The button will close the popup automatically for you.

Popups have a lot of configuration options that aren't used here, but you may find useful from the documentation. For example, you can change the OK or Cancel button text, change

the button color style, or even create a more complex popup where you define all of the buttons and properties yourself.

This concludes the features we'll build into this app. I'd like to challenge you to improve this app with some of the knowledge you've gained from earlier chapters.

6.10 Chapter challenges

This chapter covered the remaining major Ionic components. Now you've seen them in action, I challenge you to extend this weather app with additional features from earlier chapters.

- **Add a way to reload the forecast.** The current conditions are only loaded when the view is loaded. You can implement reloading of the forecast without leaving the view using the ionRefresh component or another method of your own design.
- **Implement the loading component.** The weather view and search views load data from an API, and while this is happening the user can still interact with the screen. Implement a loading component to provide feedback that the UI is waiting for the data.
- **Allow reordering of locations.** You can delete locations in the settings view, but you can also implement a way to handle reordering of the locations using the ionList reordering feature.
- **Use tabs instead of ionScroll.** You could replace the ionScroll feature with tabs for the 3 different views. Tip: Just try to use the tabs without declaring each tab as a new state.
- **Set default view to primary location.** Right now the default view is the search view, but if there is already a location stored it might be nice to view it instead.
- **Persist the favorites and settings.** So far every load of the app will reset the settings and favorites. Look at how to implement persistence in chapter 7 and improve the experience by remembering settings and favorites.

6.11 Summary

Through the last three chapters we've used most of the Ionic components and features available to us. In this chapter specifically we learned how to do the following things.

- Setup and used a side menu as a base for our navigation.
- Created a custom scrolling paged experience using ionScroll.
- Used an action sheet to display options for the user contextual to the current view.
- Created a modal for showing related information without clearing the current view.
- Learned about performance improvements and usage for collection repeat over ngRepeat.
- Added popups for giving our users feedback and confirming their actions.

7 Advanced techniques for professional apps

In this chapter you will learn how to:

- Build a customized version of the Ionic styles and use SASS.
- Handle gestures and events.
- Store and persist data between app uses.
- Modify your app to adapt to different platforms.
- Configure Ionic default behaviors and settings.

This chapter is focused on some advanced techniques that we could incorporate into most apps. As Ionic developers dig deeper into the platform, they will likely discover that the core components, while useful, cannot provide everything well designed apps will need. There should be an element of uniqueness for all apps. Just using Ionic's components out of the box without any customization or creativity is not the best approach for quality apps.

Using these various techniques, we'll be able to design apps that take the strengths of Ionic and extend it for a unique experience. You'll be able to mold the app to adapt its design and behavior for different platforms and improve the user experience through events and using storage.

7.1 Setup chapter project

This chapter is a bit different from the previous chapters where we built a full scale app. Here the examples are minimized to focus on just the parts we're discussing at the time. You can either download the chapter examples or use git to check out a copy for yourself.

The examples are organized into folders for each sample. I'll let you know which folder to look at when we are working with it. For each, you should only have to use `ionic serve` from

that folder to preview the app in the browser. For some, you might consider running it in an emulator or on a device.

7.1.1 Get the code

To get the latest copy of the chapter example, you can download the completed files or checkout the repository using Git. This link will start to download the chapter example as a zip file, which you can extract and then view. https://github.com/ionic-in-action/chapter7/archive/master.zip. To checkout the chapter examples, use the following command to clone the repository. This chapter uses the master, and doesn't have tags to checkout for each step.

```
$ git clone https://github.com/ionic-in-action/chapter7.git
```

7.2 Custom Ionic styling using SASS

Ionic comes with a beautiful set of default colors and styles for every component. The examples so far have used very little custom styling and relied heavily on Ionic's defaults. This is great for learning and shows the power of Ionic, but typically we will want to customize the design for our needs in some way.

It is best practice to customize the display of the app for your own needs. This is particularly true of the colors, since you want to give your app its own design and branding. This usually takes some time to consider what works best for your app, and boils down to your vision for the app branding and styling.

I want to reiterate that you should not try to modify the default Ionic CSS file. This is bad practice, and will cause problems when you want to update Ionic. Also, if you try to add new rules to change Ionic styling it might cause you stress long term to maintain the list, especially if you are trying to change the default colors on every component.

In this section we'll be using the example inside of the `sass` directory of our code project. You can refer to it for a working example of how SASS is configured. Let's use SASS to customize Ionic's styling for our own purposes.

7.2.1 Setting up SASS

SASS is a CSS preprocessor. SASS is superset of CSS, which means you can write regular CSS and SASS understands it. SASS compiles down to CSS, so there is nothing special that the browser needs. However, SASS provides a number of features that CSS does not (such as variables, nesting, and inheritance), which are very useful for us to customize the styling. You can learn more about SASS at http://sass-lang.com/.

Ionic has written its styling using SASS, and has used variables extensively. These variables can be declared once and used in multiple places. This makes it possible to change the variable for a color once, and have the color update anywhere it was used. There are hundreds of variables that control the primary color styles, fonts, padding, borders, and more. We can override any of the variables, and then regenerate the CSS with our new values.

First we need to setup our app to be ready for SASS. We need to make sure we've installed our node dependencies for the project, and then run the Ionic setup command which will update a few parts of our app.

```
$ npm install -g gulp
$ ionic setup sass
```

The first command will install Gulp, which is a build tool. Ionic uses Gulp to run tasks, such as the task to convert the SASS files into CSS. Gulp uses the `gulpfile.js` file that Ionic should have created in your project when you first began. The file is used to manage the Gulp tasks. You can modify (or may have already) the Gulp file with additional tasks that you wish to run, but by default Ionic only creates tasks related to building SASS.

The second command handles a few things for you for SASS. It will install any dependencies required to run using NPM, then check that your Gulp file has a SASS task. Assuming it finds one (it should unless you deleted it sometime) it will run the task and build the CSS for the first time. It will also add a few notes to the `ionic.project` file. Lastly, it tries to update the index.html file with a reference to the new customized compiled CSS file (`www/css/ionic.app.css`). It might fail to complete this depending on how much customization you have done and it might also remove all other CSS files from the index.html. You should verify the new file is correctly linked.

It is easiest to do this right away when you start a new project, instead of waiting to do it later. Now let's take a look at how to modify the default variables to customize Ionic.

7.2.2 Customize Ionic with SASS variables

Ionic has hundreds of default variables for different parts of the Ionic styling. The most obvious and useful to change are the default 9 colors options. The exact number of variables may change as Ionic updates, but you can find the complete list in the `www/lib/ionic/sass/_variables.scss` file. However, do not change them in that file! Just use it as a reference to find the variables you need to customize, and we will override them in another location.

To customize these variables, we are going to modify the `sass/scss/ionic.app.scss` file. Inside there are some comments, but really there are two commands. You can see them here.

```
// The path for our ionicons font files, relative to the built CSS in www/css
$ionicons-font-path: "../lib/ionic/fonts" !default;

// Include all of Ionic
@import "www/lib/ionic/scss/ionic";
```

The first is a variable that correctly links to the font icon directory, since this file is in a different place from the default files. The second is an import command, which will import the file found at `www/lib/ionic/scss/ionic.scss`, which then imports the rest of the SASS files. Any variables that we set before the import command will override the default variables, and is where we will be adding some new values. Anytime we add variables, we will need to rebuild the SASS files.

Imagine we want to change the default Ionic color to be colors set forth by Google's material design standard. We add our variables before the import command with our values to set a new default.

Listing 7.1 – SASS variables (sass/scss/ionic.app.scss)

```
$light: #FAFAFA;
$stable: #EEE;
$positive: #3F51B5;
$calm: #2196F3;
$balanced: #4CAF50;
$energized: #FFC107;
$assertive: #F44336;
$royal: #9C27B0;
$dark: #333;

// Include all of Ionic
@import "www/lib/ionic/scss/ionic";

@import "www/sass/app";
```

#1 Set default variables according to our requirements
#2 Import the Ionic library SASS files, which our variables will override existing Ionic ones
#3 Import a SASS file from our app

This will set anywhere we have used the color classes to be our new color, such as the bar-positive or tabs-positive color presets. We need to regenerate the CSS first, and we can do this by running the Gulp task.

```
$ gulp sass
```

The file should rebuild in less than a second, and update the `www/css/ionic.app.css` file with the new color preset. This is pretty awesome, but it can get annoying to remember to always rerun the gulp task every time you change some styling. There is also a watch task that will automatically rebuild anytime you save changes. It has to run in its own command line window, so it can be running continuously in the background. Open a new command line window or tab, and run the watch command.

```
$ gulp watch
```

Alternatively, when you use `ionic serve` and you setup SASS it will automatically rebuild the CSS automatically when you change the files, and then refresh the CSS in the browser without reloading, so you'll get to see your changes instantly.

Sometimes the `gulp watch` or `ionic serve` commands will hit an error and stop running. Depending on your command line it might alert you, but if you notice that changes don't seem to be appearing as you change them, verify that the command is still running correctly. Errors can happen when you have syntax errors that the system doesn't understand.

7.2.3 Using SASS for your own styling

You can use SASS for your customizations beyond just changing Ionic variables. It is a good idea to write all of your custom styling using SASS as well. There are many features you can use to help, but you can write CSS if that is what you prefer. I personally recommend it, even if you aren't sure that you'll need the extra features of SASS. At a minimum, it will tell you about syntax errors as soon as you try to save the file.

The easiest way to start is to create new files inside of the `scss` directory and write your styles there. You will need to then import them in the ionic.app.scss file, just like it imports the Ionic styles. Note, you will want to do this after importing the Ionic library. Here is an example of the syntax to import.

```
@import "customizations"
```

You can leave off the file extension if the files are named with .scss. By default, the Gulp task watches any SASS file in the scss directory, so it will start to automatically rebuild when you make changes to any of your styling as long as the watcher is running.

Now I like to keep my styles in the www directory. I've described before how I like to have my JavaScript, CSS, and HTML for the same view located in the same folder. This isn't a problem, because you can still use the ionic.app.scss file as the main app file, and then import the files from the www directory. By default, Ionic's Gulp task assumes you will put all of your SASS files in the scss directory, so the watcher task will not look at the www directory for changes. You can change this easily by updating gulpfile.js, where you see the paths.sass property defined. The property here takes an array of paths (which can include wildcards or glob patterns to match), and this example will add support to watch the www directory as well.

```
var paths = {
  sass: ['./scss/**/*.scss', './www/**/*.scss']
};
```

That is a simple little improvement that allows you to keep your styles together with the HTML and JavaScript for the view. You can organize your code however you like, but it is best to keep it consistent.

7.3 Handling gesture events in Ionic

Sometimes you will need to build your own component or interface and you will want to handle user gestures and events such as swipes and drags. Ionic has several options for you to use to build this support.

Very few apps can be built without creating customized interface elements. There are some apps that require very unique touch experiences to interact with the elements. I advise against creating complex gestures or relying on users learning specific gestures, because users have a low threshold for learning an app. If your custom interface doesn't make sense or provide enough contextual information about how to use it, then users are likely to abandon your app. Nobody likes to feel dumb or confused, so be considerate of how you build these interactions by favoring simplicity.

The two main ways Ionic provides support for gestures are with a set of directives to listen for events, or adding event listeners programmatically into your controllers.

7.3.1 Listen for events with Ionic event directives

The Ionic event directives are a collection that will listen for a particular event and call an expression or function when the event fires. Some of the events include on hold, tap, drag, or swipe. The exact timing for these events to fire are listed in the documentation for these events. This section uses the `events` directory of our project, and figure 7.2 shows the output.

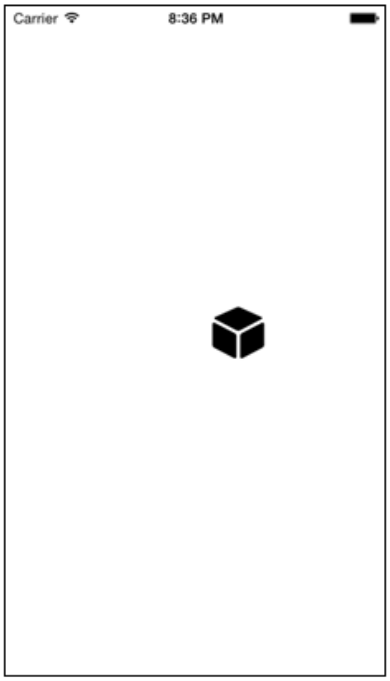


Figure 7.2 – Draggable box using event directives

These event directives are the easiest way to listen for events. Let's take a look at a an example of how to use them. I've created a directive that has a combination of events that lets you drag an icon around the screen, and it will also log to the console the number of milliseconds from the time the user touched the item until they released it. The code in listing 7.2 has the directive, which for simplicity I have added into the app.js file.

Listing 7.2 – Draggable box directive (events/www/js/app.js)

```
angular.module('App', ['ionic'])
.directive('box', function () {
  return {
```

```
    link: function (scope, element) { #1
      var time = 0, boxX = 0, boxY = 0; #2
      var leftBound = window.innerWidth - 50; #2
      var bottomBound = window.innerHeight - 50; #2
      scope.top = 0; #2
      scope.left = 0; #2

      scope.startTouch = function (event) { #3
        time = event.timeStamp; #3
      }; #3
      scope.endTouch = function (event) { #4
        console.log('You held the box for ' + (event.timeStamp - time) + 'ms'); #4
        boxX = scope.left; #4
        boxY = scope.top; #4
      }; #4
      scope.drag = function (event) { #5
        var left = boxX + Math.round(event.gesture.deltaX); #5
        var top = boxY + Math.round(event.gesture.deltaY); #5

        if (left > leftBound) { #5
          scope.left = leftBound; #5
        } else if (left < 0) { #5
          scope.left = 0; #5
        } else { #5
          scope.left = left; #5
        } #5
        if (top > bottomBound) { #5
          scope.top = bottomBound; #5
        } else if (top < 0) { #5
          scope.top = 0; #5
        } else { #5
          scope.top = top; #5
        } #5
      }; #5
    },
    template: '<div id="box" class="icon ion-cube" on-touch="startTouch($event)" on-
      release="endTouch($event)" on-drag="drag($event)" ng-style="{top: top + \'px\',
      left: left + \'px\'}"></div>' #6
  }
})
```

#1 Link function for our box directive to add listeners
#2 Setup some variables to track positions
#3 Touch event handler, track the start time of drag
#4 Release event handler, track the total time of drag and log to console
#5 Drag event handler, moves the position of the box based on drag, limits boundaries to edge
#6 Inline template, box is an icon with events and styles updated by drag

In this example, we've created an icon that can be moved around the screen. It does check that the icon does not go outside of the window space, otherwise it will go anywhere the user drags it. The `onTouch` and `onRelease` event handlers are used to track the total time the user touches the icon, and the `onDrag` event handler does the work to move it around by changing scope variables for the top and left position that `ngStyle` updates.

To use this, we just add a box element to the app. Here we just add the single box to the app and it will allow us to begin dragging.

```
<body ng-app="App">
  <box></box>
</body>
```

There is also some CSS required for the positioning to work. CSS rules allow us to position an element absolutely by giving it a top and left position value, which we get from the drag event.

```
#box {
  position: absolute;
  width: 50px;
  height: 50px;
  font-size: 50px;
  text-align: center;
}
```

There are many different ways you could accomplish similar tasks, but I wanted to highlight how the event directives can be used to react to user gestures. I placed this example into a directive because best practice is to use a directive when you need to manipulate the DOM. However you could also have used the directives in a more standard template and put the event handler methods onto the controller.

7.3.2 Listen for events with Ionic gestures service

Another way to listen for events is to use the `$ionicGesture` service. This allows you to listen for a wider range of events, but requires a more programmatic approach. The example for this section is found in the `gestures` directory of the project.

The `$ionicGesture` service needs to be injected into the controller, and then you can declare which events to listen to. You also have to declare the element to which the listener will attach itself. This makes it even more important to use the gestures service inside of a directive when possible, so you have easy access to the element.

Figure 7.3 – Gesture event listeners to swipe cards off the screen

We'll use the service to build a simple card that can be swiped off the screen like you see in figure 7.3. While the user swipes the card to the right or left it will animate in that direction, and once they release if the card is far enough it will be removed from the screen or it will reset to the center. This is found in the chapter 7 project inside of the gestures directory.

Listing 7.x – Ionic gestures service (gestures/www/js/app.js)

```
angular.module('App', ['ionic'])
.directive('card', function () {
  return {
    scope: true,
    controller: function ($scope, $element, $ionicGesture, $interval) { #1
      $scope.left = 0;

      $ionicGesture.on('drag', function (event) { #2
        $scope.left = event.gesture.deltaX; #2
        $scope.$digest(); #2
      }, $element); #2

      $ionicGesture.on('dragend', function (event) { #3
        if (Math.abs($scope.left) > (window.innerWidth / 3)) { #4
          $scope.left = ($scope.left < 0) ? -window.innerWidth : window.innerWidth; #4
          $element.remove(); #4
        } else { #5
          var interval = $interval(function () { #5
```

```
            if ($scope.left < 5 && $scope.left > -5) { #5
              $scope.left = 0; #5
              $interval.cancel(interval); #5
            } else { #5
              $scope.left = ($scope.left < 0) ? $scope.left + 5 : $scope.left - 5; #5
            } #5
          }, 5); #5
        } #5
        $scope.$digest(); #5
      }, $element);
    },
    transclude: true,
    template: '<div class="list card" ng-style="{left: left + \'px\'}"><div
      class="item" ng-transclude>Swipe Me</div></div>'
  }
})
```

#1 Inject the $ionicGesture service into controller
#2 Listen for drag event, and move the card horizontally while card is dragging
#3 Listen for dragend event, and determine if card should be removed or reset
#4 If the card is over 33% of the way off the screen, remove it
#5 If the card is still near the middle, animate it back to center by moving 5px every 5 milliseconds

This card directive attach two event listeners for `drag` and `dragend`. Technically we are listening for the drag events here, because swipe events do not fire until the swipe has occurred. If we listened to the swipe event, the cards would not move until after the user had already completed the swipe, so it would have a visual delay that might confuse the user. We use the controller of the directive because we need to inject the service. When we attach an event listener using the on method, we have to pass at least three things. First is to declare what the event name, then a callback function to fire when the event is triggered, and lastly the element it should attach to. Since we are using a directive here, we have access to the special `$element` service, otherwise in a controller you would have to use `angular.element()` locate the proper element to attach the listener.

For this example to work, we just need to add one line of CSS and add any number of cards to our app.

```
.card { position: relative; }
```

Then we can add any number of these card directives to our app, and each can be individually swiped off the screen.

```
<body ng-app="App">
  <card>Card 1</card>
     <card>Card 2</card>
  <card>Card 3</card>
  <card>Card 4</card>
  <card>Card 5</card>
</body>
```

The contents of the card element are transcluded inside of the card, which is an Angular feature available to directives. It essentially will copy all of the HTML content inside of the directive, and place it into the directive template where `ngTransclude` is declared.

This approach is more flexible and is able to support more gesture events than the event directives we looked at earlier. However, they require a little more work to setup. In the end, both of them accomplish the same task so the choice ultimately is preference over style.

7.3.3 Available gesture events

There are a lot of gesture events that we can listen for. The following table 7.1 gives a list of the possible gestures, the event name, the directive (if available), and notes about what triggers the gesture event.

Table 7.1 - List of supported gestures, JavaScript event names, usage notes, and possible directive if available.

Gesture	Event	Directive	Notes
Hold	hold	on-hold	Touch an element for at least 500ms.
Tap	tap	on-tap	Touch an element for less than 250ms.
Double Tap	doubletap		Two touches on same place, within 300ms
Touch	touch	on-touch	Fires when a touch is detected
Release	release	on-release	Fires when a touch is released
Drag	drag	on-drag	Long touch while moving in any direction, generic
Drag Start	dragstart		Fires when drag is first detected
Drag End	dragend		Fires when drag is released
Drag Up	dragup	on-drag-up	Drag up on Y axis
Drag Down	dragdown	on-drag-down	Drag down on Y axis
Drag Left	dragleft	on-drag-left	Drag left on X axis
Drag Right	dragright	on-drag-right	Drag right on X axis
Swipe	swipe	on-swipe	Quick touch and flick in any direction, generic
Swipe Up	swipeup	on-swipe-up	Swipe up on Y axis
Swipe Down	swipedown	on-swipe-down	Swipe down on Y axis
Swipe Left	swipeleft	on-swipe-left	Swipe left on X axis
Swipe Right	swiperight	on-swipe-right	Swipe right on X axis
Transform	transform		Two finger touch and move, generic
Transform Start	transformstart		Fires when a transform is first detected
Transform End	transformend		Fires when a transform is released

Rotate	rotate	Two fingers rotating
Pinch	pinch	Two fingers pinch and slide together or apart
Pinch In	pinchin	Two fingers pinch and slide together
Pinch Out	pinchout	Two fingers pinch and slide apart

7.4 Storing data for persistence

In our examples from chapters 4-6, every time we loaded the app it would reset any changes we had made to our app and start as if it was the first time the app was used. This is obviously annoying and a bad experience for our users. For example, we expect that if we favorite an item that it will stay favorite. Wouldn't it be great if our app could remember things and pick up where it left off? The good news is there are many different ways to do this, and I'll show you two primary ways that don't require any additional plugins.

Since we are building web applications, we have the ability to use some of the built in storage features of the web platform, and with Cordova we have support for localStorage for key value pairs and either WebSQL or IndexedDB for a more robust database.

The general approach for either option is that you will need to store data, and when the app resumes the first task will be to load the data from storage. Any app that has the ability to login will retain some kind of session and user information in storage to properly communicate with a backend service.

Apps with persistent data should also be designed to handle the situation where data is cleared from the cache. Never assume that stored data will remain indefinitely.

When you store data on the device, you should take precautions against storing anything that the user shouldn't be able to see. Anything on a device is potentially viewable by the device owner using debugging tools, but it is reasonable to store private data for that user (such as an oAuth token). Anything that should never be shown to the user should be stored in a server environment (such as a private API key for a web service).

7.4.1 Using local storage (localStorage)

Local storage is a very simple storage option for your app which stores values in the browser cache directory. It is essentially a key value pair storage system, or you can think of it like a JavaScript object with the ability to create only one level of properties. I personally turn to local storage anytime I can, since it is the easiest way to store data. In a browser, users can clear the data from local storage anytime, but in a hybrid app they are not able to clear the data unless they are using debugging tools.

It is very easy to use but has two major limitations. First, values are stored as a string, regardless of what data type it was before. This means an integer will be turned into a string during storage. This can cause problems if you try to compare a string to a number using strict comparators (such as `"1" === 1` which is false). Second, there are size limits on the total data you can store, which is not standardized between browsers. You should consult the documentation for the platforms you build to see what the current capacity is (Android Browser

4.3 has 2MB, Safari has 5MB, and Chrome has 10MB as of writing) or visit http://www.html5rocks.com/en/tutorials/offline/quota-research/ for a good summary of many storage type limitations. This is a lot of space, but if you exceed it you will get errors. It can become difficult to manage over time if you are working with a lot of data.

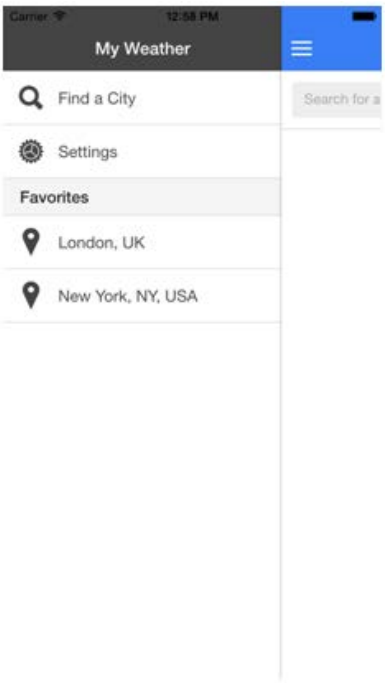


Figure 7.4 – Local storage stores the location list and reloads it when the app is opened

However, if your storage needs are simple, local storage is often the best solution. I've updated the weather app example from chapter 6 to store the locations list in local storage and load it back into the app when we reload, as shown in figure 7.4. For this section we'll use the code inside of the `storage` directory. The only change from the chapter 6 example is in the Locations service in the storage/www/js/app.js file, shown in listing 7.3 and with changes in bold.

Listing 7.3 – Save and load from localStorage (storage/www/js/app.js)

```
.factory('Locations', function ($ionicPopup) {
  function store () { #1
    localStorage.setItem('locations', angular.toJson(Locations.data)); #1
  } #1

  var Locations = {
    data: [],
```

```
    getIndex: function (item) {
      var index = -1;
      angular.forEach(Locations.data, function (location, i) {
        if (item.lat == location.lat && item.lng == location.lng) {
          index = i;
        }
      });
      return index;
    },
    toggle: function (item) {
      var index = Locations.getIndex(item);
      if (index >= 0) {
        $ionicPopup.confirm({
          title: 'Are you sure?',
          template: 'This will remove ' + Locations.data[index].city
        }).then(function (res) {
          if (res) {
            Locations.data.splice(index, 1);
          }
        });
      } else {
        Locations.data.push(item);
        $ionicPopup.alert({
          title: 'Location saved'
        });
      }
      store(); #2
    },
    primary: function (item) {
      var index = Locations.getIndex(item);
      if (index >= 0) {
        Locations.data.splice(index, 1);
        Locations.data.splice(0, 0, item);
      } else {
        Locations.data.unshift(item);
      }
      store(); #2
    }
  };

  try { #4
    var items = angular.fromJson(localStorage.getItem('locations')) || [];
    Locations.data = items;
  } catch (e) {
    Locations.data = [];
  }

  return Locations;
})
```

#1 Create a store method, to handle saving JSON string data into local storage
#2 Call the store method after toggling a location from the list
#3 Call the store method after setting a new primary location
#4 When the app starts, try to load data from local storage or else set a blank array

You can see local storage is available to us globally in JavaScript, because it is part of the primary JavaScript APIs. In this example I created a `store()` function to abstract the logic, since there are multiple places we want to store the data into local storage. When `store()` is

called, it takes the array of locations and stores them into local storage, but first converts the array into a JSON string (since local storage only handles strings). Then when the list of locations changes, we just call `store()` to update the cached data.

In the try/catch, we attempt to load the data from local storage, and then parse the JSON if it was set. If nothing is set in local storage or if there is an error loading data from local storage, then the location list is set to an empty array.

Now our app will always try to load the list of stored locations and use that instead of starting with a blank list. This is obviously a very good improvement for our users, and using local storage is very easy to implement.

You can inspect the local storage values inside of the browser developer tools and should see an item with the list there. Local storage is app specific, so the data you store is safe from other apps. However, local storage can be inspected by developers, meaning you cannot safely store anything that you absolutely cannot allow others to see.

7.4.2 Using WebSQL/IndexedDB

WebSQL and IndexedDB are two types of browser based databases. Like local storage, the data is stored in the browser's cache system. These options are best for larger amounts of data, or data that you will want to be able to directly query. However, they are both more difficult to use and support for them varies across platforms.

WebSQL is similar to a full featured database with the ability to use SQL to query tables. It allows you to use SQL statements like SELECT, UPDATE, and so forth. The challenge is the specification for WebSQL was abandoned back in 2010 when browser vendors could not come to agreement on the standard. At the time of writing, iOS and Android both support WebSQL, but it is possible that over time this may be removed.

IndexedDB is an object store, which is somewhere between WebSQL and local storage. It also uses a key value storage, like local storage, but items have fields with specific data types and the ability to limit results by requesting certain fields with a given value. At the time of writing, IndexedDB is not supported by iOS and Android.

Like local storage, WebSQL or IndexedDB are cannot be viewed by other apps, but can still be viewed by developers when they debug using their device.

At the time of writing, WebSQL is the option that both iOS and Android support fully with the help of Cordova, and IndexedDB is not properly supported. However, WebSQL has been deprecated since 2010 and likely will be removed in the future, so in time support will likely shift to IndexedDB. However, to be doubly sure what is supported or not, you can check the Cordova storage documentation http://cordova.apache.org/docs/en/edge/cordova_storage_storage.md.html#Storage. Check that you are looking at the same version of the documentation that fits the project's Cordova version, and you can run `cordova info` to find the version. You could also do a quick test yourself by running one of the following to tell you if it is supported or not on a given platform. You do need to run these on an emulator or device to get the proper message. Just add this at the start of your JavaScript to alert you for support for the two options.

```
alert('WebSQL: ' + ((window.openDatabase) ? 'yes' : 'no'));
alert('IndexedDB: ' + ((window.indexedDB) ? 'yes' : 'no'));
```

7.4.3 Other options from Cordova plugins

Cordova provides plugins to allow you to access additional features on the device. We will look at some plugins in chapter 8 in depth, but you should know there are many options in the Cordova plugin repository for storage.

The options are varied and ever changing. Some are able to bring IndexedDB or WebSQL support to all devices, others support different storage systems like SQLite, and others are designed to allow you to store entire files. You can discover storage plugins at http://plugins.cordova.io/.

7.5 Building one app for multiple platforms

One of the best features of building apps with Ionic is the ability to target multiple devices and platforms with one app. However, there are times that we need to tweak behavior or design for a particular device or platform.

There are different situations where we need to think about different experiences for different platforms. Ionic provides some of this built into its core. For example, tabs on Android appear differently than they do on iOS. The reason is the Ionic developers wanted to be able to provide the same behavior (tabs) but make it look and feel native to that platform (styling). The tabs do the same thing, just the appearance varies slightly.

There are two main ways to target a platform, change the appearance and/or change the behavior. Before we look at them, let's dig a bit more into why we should bother building apps that mold to different platforms.

7.5.1 One size does not always fit all

As app developers, we should be considerate of what makes the best app for our users, not what makes the app the easiest for us to build. Building apps with the exact same behavior on Android and iOS may not always work out for our users, and we should consider this carefully. This is especially true in cases where users are accustomed to certain interactions.

Android and iOS have many differences in their appearance and interaction behaviors. Even different versions of iOS and Android can differ greatly, and over time we can only assume that will continue. Ionic is committed to supporting the modern versions of these platforms, and as the mobile platforms continue to evolve Ionic will adapt.

We must remember that Ionic is only able to do so much for app developers. Ultimately we are responsible for ensuring that the apps we build work and make sense on different platforms. It is worth spending time with the official native style guides for iOS and Android to familiarize yourself with the differences. Then when you are designing your app, you will be able to consider the best design for each platform and if you need to design anything specific to a platform.

- Android Style Guide http://developer.android.com/design/style/index.html
- iOS Style Guide

https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/

7.5.2 Adapt styling for a platform or device type

Ionic provides us a simple way to determine which platform or device we are using so we can adapt our app styling as necessary. Ionic determines what platform you are using, and adds a number of classes to the body element.

- `platform-ios` for iOS
- `platform-android` for Android
- `platform-browser` for browsers

These classes give us insight into what type of platform is used. You can also find other classes based on the version number of the platform, for example `platform-ios-ios7`. In some cases you might need to target a specific version, so the version class can provide you that information.

The two major reasons you will need to use this technique are for providing platform specific styling, and to address possible display bugs present only on a particular platform. In general, you probably will want to limit the amount of platform specific design, because it will add to the amount of testing you need to do.

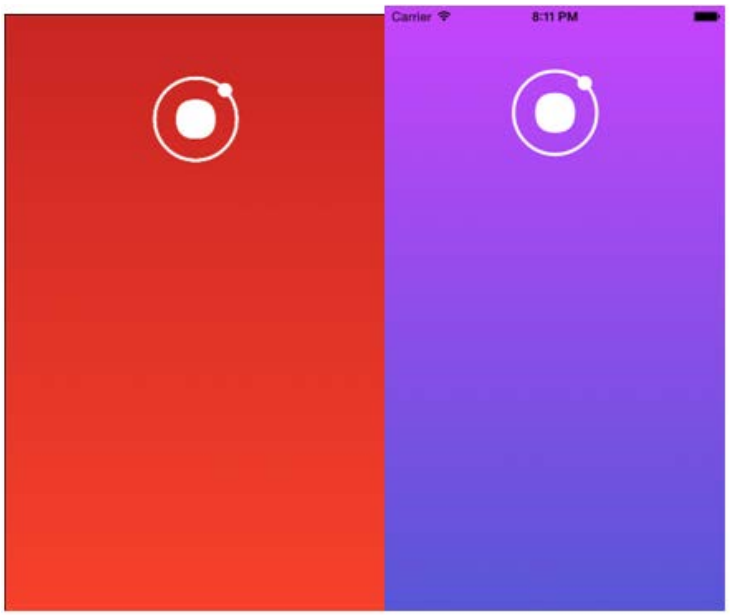


Figure 7.5 – Android with red colors on left, iOS with purple colors on right

We'll use the code from the `adaptive-behavior` directory for this section. We have a simple app that just shows the Ionic logo on a background, but depending on the platform a different background color will display. The template for our app is in listing 7.5 and the CSS in listing 7.6.

Listing 7.5 – Adaptive styling template (adaptive-style/www/index.html)

```
<body ng-app="App">
  <ion-pane>
    <ion-content>
      <span class="icon ion-ionic"></span>
    </ion-content>
  </ion-pane>
</body>
```

Listing 7.6 – Adaptive styling CSS (adaptive-style/www/css/style.css)

```
.scroll {
  text-align: center;
  padding-top: 50px;
}
.ion-ionic {
  font-size: 100px;
  color: #fff;
}
.pane {
  background: #333;
}
.platform-ios .pane { #1
  background: #C644FC;
  background: -webkit-linear-gradient(top, #C644FC 0%,#5856D6 100%);
  background: linear-gradient(to bottom, #C644FC 0%,#5856D6 100%);
}
.platform-android .pane { #2
  background: #C62828;
  background: -webkit-linear-gradient(top, #C62828 0%,#F44336 100%);
  background: linear-gradient(to bottom, #C62828 0%,#F44336 100%);
}
```

#1 CSS selector to target iOS only
#2 CSS selector to target Android only

By prefixing our CSS rules with the platform body class, we can see the different colors in the background by platform.

7.5.3 Adapt behavior for a platform or device type

We can also adapt the behaviors of our app for a particular platform. For example, you may want to use an action sheet component on iOS but a popover on Android to better fit in with the platform. Ionic allows us to detect which platform we are using, and then use that to modify behaviors.

The `ionic.Platform` service is available to provide us this information. It provides a list of methods such as isIOS() and isAndroid() to return a boolean value if the platform is active or you can also use platform() to return the name of the current platform.

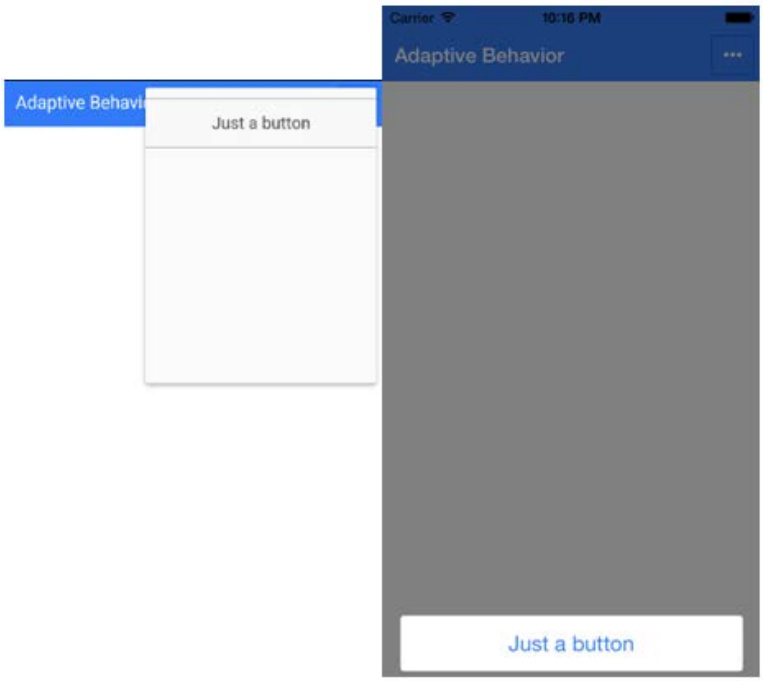


Figure 7.6 – Based on platform, change the behavior of the button in iOS or Android

In a fairly simple example, shown in figure 7.6, the more button changes behavior depending on the platform. We'll check if its iOS, and show the action sheet. Otherwise, we show the popover for Android.

Listing 7.7 – Adapting behavior by platform (adaptive-behavior/www/js/app.js)

```
angular.module('App', ['ionic'])
.controller('Controller', function ($scope, $ionicActionSheet, $ionicPopover) { #1
  $scope.more = function (event) { #2
    if (ionic.Platform.isIOS()) { #3
      $ionicActionSheet.show({ #4
        buttons: [ #4
          {text: 'Just a button'} #4
        ], #4
        buttonClicked: function (index) { #4
          return true; #4
        } #4
      }); #4
    } else { #5
      var popover = $ionicPopover.fromTemplate('<ion-popover-view><button
       class="button button-full">Just a button</button></ion-popover-view>'); #5
      popover.show(event); #5
    } #5
  }
})
```

#1 Create the controller and inject services
#2 More method to be called by ngClick
#3 Use ionic.Platform to determine if iOS
#4 If iOS, show action sheet with dummy button
#5 Otherwise, show popover with dummy button

Here we've created a controller with a single method that checks if the device is running iOS or not. The `ionic.Platform` service is not an Angular service, so we don't need to inject it. There is an `$ionicPlatform` service, but it is intended for use with Cordova plugins and does not provide information about the current platform.

Once we've determined the platform, we decide to show the action sheet or popover. The markup for this example is in listing 7.8.

Listing 7.8 – Adaptive behavior template (adaptive-behavior/www/index.html)

```
<body ng-app="App">
  <ion-header-bar align-title="left" class="bar-positive" ng-controller="Controller">
    <h1 class="title">Adaptive Behavior</h1>
    <div class="buttons">
      <button class="button" ng-click="more($event)"><span class="icon ion-
      more"></span></button> #1
    </div>
  </ion-header-bar>
</body>
```

#1 Use ngClick to call the more method, and pass the event for the popover

The `ionic.Platform` service is able to provide us with current information about the platform. It also has a few methods to modify the app behavior, such as the ability to set the app to full screen or exit the app programmatically.

7.6 Modify default behaviors with $ionicConfigProvider

Ionic has a way to modify a number of the default behaviors. We were able to modify the default styling using our own SASS variables, and this is the same idea except we can modify behaviors such as transition types or default navbar title alignment.

The defaults for Ionic are designed to be focused on the correct platform. For example, in a navbar the title will align to the left on Android and will center on iOS to match the style guidelines. However, you can force Ionic to render the titles the same regardless of the platform.

The complete list of configurable items is provided on the documentation, and we'll build one example to modify the default tabs styling so that it is always striped, and we want the tabs on top. All of the configuration options can be modified in the same manner as you see in this example. In figure 7.7 you can see the result of updated defaults for the tabs.

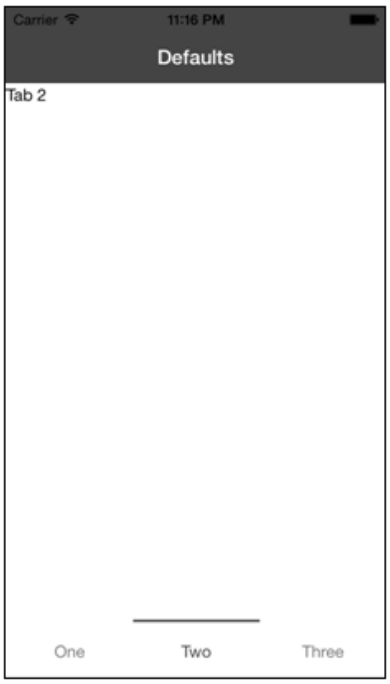


Figure 7.7 – Overriding Ionic default values for tabs

The configuration defaults are set in the module config method, in the same method that the states are declared. Listing 7.9 has the default configuration changes set for tabs.

Listing 7.9 – Updating default config (config/www/js/app.js)

```
angular.module('App', ['ionic'])
.config(function($ionicConfigProvider) { #1
  $ionicConfigProvider.tabs.style('striped').position('bottom'); #2
})
```

#1 Inject $ionicConfigProvider into config method
#2 Call tabs settings, and they can be chained in some cases

The $ionicConfigProvider is the special service provider for Ionic's configuration, and we are able to update values by calling its methods and passing arguments. In this example, we also can chain the two tabs methods together, but if we were changing the default for another aspect unrelated to tabs it would not work. This will set tabs to be striped and on the bottom, which is not the default behavior for tabs.

The configurations can still be overridden in the tabs implementation using classes. It might not be necessary to change the default for some things like the tabs display, since you can still

set the CSS classes on the tabs instance to modify its display. However, some of the configurations cannot be changed elsewhere, particularly the caching views information.

7.7 Summary

This chapter has given us additional tools and insights into how to build Ionic apps. We covered the following concepts.

- Learned about how to build a custom version of the Ionic styles for our app using SASS, and about the build processes that Ionic uses with Gulp.
- Added support for events and gestures, both using the event directives and the $ionicGesture service.
- Utilized local storage to persist data in the app, and also looked at other options such as WebSQL/IndexedDB.
- Modified the app behavior and display based on the current platform of the device running the app to provide specialized experiences per platform.
- Changed the default Ionic configuration to set global parameters for different parts of Ionic.