



Build Mobile Apps with Ionic 2 and Firebase

Hybrid Mobile App Development

Fu Cheng

Apress®

Build Mobile Apps with Ionic 2 and Firebase

Hybrid Mobile App Development



Fu Cheng

Apress®

Build Mobile Apps with Ionic 2 and Firebase: Hybrid Mobile App Development

Fu Cheng
Sandringham, Auckland
New Zealand

ISBN-13 (pbk): 978-1-4842-2736-7
DOI 10.1007/978-1-4842-2737-4

ISBN-13 (electronic): 978-1-4842-2737-4

Library of Congress Control Number: 2017941053

Copyright © 2017 by Fu Cheng

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Technical Reviewer: Massimo Nardone
Coordinating Editor: Jessica Vakili
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-2736-7. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper



To my wife Andrea and my daughter Olivia



Contents at a Glance

About the Author	xv
About the Technical Reviewer	xvii
Preface	xix
■ Chapter 1: Getting Started	1
■ Chapter 2: Languages, Frameworks, Libraries, and Tools	19
■ Chapter 3: Basic App Structure	47
■ Chapter 4: List Stories	57
■ Chapter 5: View Story	117
■ Chapter 6: View Comments	127
■ Chapter 7: User Management	139
■ Chapter 8: Manage Favorites	177
■ Chapter 9: Share Stories	195
■ Chapter 10: Common Components	203

■ **Chapter 11: Advanced Topics 215**

■ **Chapter 12: End-to-End Test and Build..... 227**

■ **Chapter 13: Publish 239**

Index..... 247

Contents

About the Author	xv
About the Technical Reviewer	xvii
Preface	xix
■ Chapter 1: Getting Started	1
Mobile Apps Refresher	1
Hybrid Mobile Apps	2
Apache Cordova.....	3
Ionic Framework.....	3
Firebase.....	4
Prepare Your Local Development Environment	5
Node.js.....	5
Ionic CLI	5
iOS	7
Android	7
IDEs and Editors	9
Create an App Skeleton.....	9
Blank App	9
Tabbed App	10

Sidemenu	11
Tutorial.....	12
Local Development.....	13
Use Chrome for Development.....	14
Use Chrome DevTools for Android Remote Debugging	15
Test on Emulators.....	15
iOS	15
Android	16
Summary	17
■ Chapter 2: Languages, Frameworks, Libraries, and Tools	19
TypeScript	20
Why TypeScript?	20
Basic Types.....	21
Functions	25
Interfaces and Classes	27
Decorators	31
Angular 2	34
RxJS	41
Observable.....	41
Observers	42
Subjects.....	42
Operators.....	43
Sass.....	43
Variables.....	43
Nesting	44
Mixins	44
Jasmine and Karma	45
Summary.....	45

Chapter 3: Basic App Structure	47
Understanding the Basic App Structure	48
Config Files.....	48
package.json.....	48
config.xml.....	49
tsconfig.json	49
ionic.config.json.....	50
tslint.json	50
.editorconfig	50
Cordova Files.....	50
hooks.....	50
platforms	50
plugins.....	51
www	51
App Files.....	51
index.html.....	51
declarations.d.ts.....	51
manifest.json and service-worker.js	51
assets	51
theme	52
app.....	52
components.....	52
pages.....	52
Skeleton Code	52
app.module.ts.....	52
app.component.ts.....	53
app.html.....	54

main.ts.....	55
app.scss.....	55
Page 1 and Page 2 Files	55
Summary	55
Chapter 4: List Stories	57
Define the Model	57
List Component	58
Simple List.....	58
Header and Separators.....	59
Grouping of Items	59
Icons	60
Avatars.....	61
Thumbnails	61
Display a List of Items	62
Item Component	63
Items Component	65
Empty List.....	65
Test List Component.....	66
Tools	66
Testing Configuration.....	67
Testing Items Component	71
Run Tests	75
Items Loading Service.....	76
Top Stories Page.....	78
Test	79
Firebase Basics	82
Database Structure.....	82
Firebase JavaScript SDK	83
Write Data.....	86

Query Data.....	87
Navigation.....	89
Hacker News API	89
AngularFire2	90
API	91
Implement ItemService	92
Manage the Changes.....	93
Further Improvements	95
Pagination and Refresh	98
Advanced List	102
Customization.....	105
Testing	107
Loading and Error.....	108
Loading.....	108
Error Handling.....	112
Summary	116
■ Chapter 5: View Story.....	117
In App Browser	117
Installation	118
Open a URL	118
Alerts	121
A Better Solution.....	122
Styles.....	125
Testing	126
Summary	126
■ Chapter 6: View Comments	127
Navigation	127
Basic Usage	127
Page Navigation.....	128

Model	128
Refactoring.....	129
Services.....	129
Pages.....	131
View Comments	133
CommentComponent.....	133
CommentsComponent	134
Items.....	135
View Comments.....	135
CommentsPage	136
Summary	138
Chapter 7: User Management	139
Ionic UI Controls	139
Inputs.....	140
Check Box.....	140
Radio Buttons	141
Selects.....	142
Toggles	144
Ranges.....	144
Labels	146
Modal.....	146
Toolbar	147
Menu.....	148
Email and Password Login	150
Model.....	150
AuthService	150
Sign-Up Form	154
Login Page.....	158
Menu.....	160

Third-Party Login.....	163
Google Login.....	164
Facebook Login	168
GitHub Login	170
Test.....	174
Summary	175
■ Chapter 8: Manage Favorites.....	177
Favorites Service.....	177
Favorites Page.....	180
Favorites Items.....	184
Testing.....	187
Summary	193
■ Chapter 9: Share Stories.....	195
Card Layout	195
Grid Layout	196
Sharing.....	198
More About the Plugin	200
Summary	201
■ Chapter 10: Common Components	203
Action Sheet.....	203
Popover	206
Slides.....	209
Tabs	212
Summary	214

■ Chapter 11: Advanced Topics	215
Platform.....	215
Theming	216
Colors	218
Config	219
Storage.....	219
Push Notifications	221
Summary.....	225
■ Chapter 12: End-to-End Test and Build.....	227
End-to-End Test with Protractor	227
Protractor Config	228
Top Stories Page Test.....	230
Page Objects and Suites.....	231
User Management Test.....	233
Favorites Page Test.....	234
Build	236
PhantomJS for Unit Tests.....	236
Gitlab CI	236
Summary	237
■ Chapter 13: Publish	239
Icons and Splash Screens	239
Deploy to Devices.....	240
View and Share with Ionic View	241
Ionic Deploy.....	242
Cloud Client	242
Deploy Service	243
Summary.....	246
Index.....	247



About the Author

Fu Cheng is a full-stack software developer working in a healthcare start-up in Auckland, New Zealand. During his many years of experience, he worked in different companies to build large-scale enterprise systems, government projects, and SaaS products. He is an experienced JavaScript and Java developer and always wants to learn new things. He enjoys sharing knowledge by writing blog posts, technical articles, and books.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in Security, Web/Mobile development, Cloud, and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

Technical skills include the following: Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015).

This book is dedicated to Antti Jalonon and his family who are always there when I need them.

Preface

Developing mobile apps is an interesting yet challenging task. Different mobile platforms have their own ecosystems. There are new programming languages, frameworks, libraries, and tools to learn. Building complicated mobile apps or games requires a lot of experience. But not all mobile apps are complicated. There are still many mobile apps that are content-centric. This kind of apps focuses on content presentations and doesn't use many native features. For these kinds of apps, PhoneGap and its successor Apache Cordova offer a different way to build them.

Mobile platforms usually have a component to render web pages. Cordova uses this component to create a wrapper for running web pages. Cordova provides different wrappers for different platforms. The web pages become the mobile apps to develop. After using Cordova, developers can use front-end skills to create cross-platform mobile apps. This empowers front-end developers to create good enough content-centric mobile apps. Many other frameworks build on top of Cordova to provide out-of-box components to make building mobile apps much easier.

This book focuses on the latest version 2 of the popular Ionic framework. The best way to learn a new framework is using it in real product development. This book is not a manual for Ionic 2, but a field guide of how to use it. We'll build a Hacker News client app using Ionic 2 and use this as the example to discuss different aspects of Ionic 2. This book not only covers the implementation of the Hacker News client app, but also the whole development life cycle, including unit tests, end-to-end tests, continuous integration, and app publish. After reading this book, you should get a whole picture of building mobile apps using Ionic 2.

Most of the nontrivial mobile apps need back-end service to work with them. Using mobile apps back-end services is a new trend that eliminates the heavy burden to write extra code and maintain the back-end infrastructure. Google Firebase is a popular choice of mobile apps back-end services. The Hacker News client app uses Firebase to handle user authentication and user favorites data storage. After reading this book, you should be able to integrate Firebase in your own apps.

Prerequisites

Ionic 2 builds on top of Angular 2 and uses TypeScript instead of JavaScript. Basic knowledge of Angular 2 and TypeScript is required to understand the code in this book. This book provides the basic introduction to Angular 2 and TypeScript, but it's still recommended to refer to other materials for more details.

To build Ionic 2 apps running on iOS platform, macOS is required to run the emulator and Xcode. You may also need real physical iOS or Android devices to test the apps.

Acknowledgments

This book would not have been possible without the help and support of many others. Thank you to my editors, Aaron Black, Jessica Vakili, and James Markham; and the rest of the Apress team, for bringing this book into the world. Thank you to my technical reviewer Massimo Nardone for your time and insightful feedback.

Many thanks to my whole family for the support during the writing of this book.

Getting Started

Mobile apps development is a hot topic for both companies and individual developers. You can use various kinds of frameworks and tools to build mobile apps for different platforms. In this book, we use Ionic 2 to build so-called hybrid mobile apps. As the first chapter, this chapter provides the basic introduction of hybrid mobile apps and helps you to set up the local environment for development, debugging, and testing.

Mobile Apps Refresher

With the prevalence of mobile devices, more and more mobile apps have been created to meet all kinds of requirements. Each mobile platform has its own ecosystem. Developers use SDKs provided by the mobile platform to create mobile apps and sell them on the app store. Revenue is shared between the developers and the platform. Table 1-1 shows the statistics of major app stores at the time of writing.

Table 1-1. *Statistics of major app stores*

App Store	Number of available apps	Downloads to date
App Store (iOS)	2.2 million	140 billion
Google Play	2.6 million	65 billion
Windows Store	669,000+	--
BlackBerry World	240,000+	4 billion
Amazon Appstore	334,000+	--

The prevalence of mobile apps also creates a great opportunity for application developers and software companies. A lot of individuals and companies make big money on the mobile apps markets. A classic example is the phenomenal mobile game Flappy Bird. Flappy Bird was developed by Vietnam-based developer Dong Nguyen. The developer claimed that Flappy Bird was earning \$50,000 a day from in-app advertisements as well as sales. Those successful stories encourage developers to create more high-quality mobile apps.

Let's now take a look at some key components of mobile app development.

Hybrid Mobile Apps

Developing mobile apps is not an easy task. If you only want to target a single mobile platform, then the effort may be relatively smaller. However, most of the times we want to distribute apps on many app stores to maximize the revenue. To build that kind of apps which can be distributed to various app stores, developers need to use different programming languages, SDKs, and tools, for example, Objective-C/Swift for iOS and Java for Android. We also need to manage different code bases with similar functionalities but implemented using different programming languages. It's hard to maximize the code reusability and reduce code duplications across different code bases, even for the biggest players in the market. That's why cross-platform mobile apps solutions, like Xamarin (<https://www.xamarin.com/>), React Native (<https://facebook.github.io/react-native/>), and RubyMotion (<http://www.rubymotion.com/>) also gain a lot of attention. All these solutions have a high learning curve for their programming languages and SDKs, which creates a burden for ordinary developers.

Comparing to Objective-C/Swift, Java, C# or Ruby, web development skills, for example, HTML, JavaScript, and CSS are much easier to learn. Building mobile apps with web development skills is made possible by HTML5. This new type of mobile apps is called hybrid mobile apps. In hybrid mobile apps, HTML, JavaScript, and CSS code run in an internal browser (WebView) that is wrapped in a native app. JavaScript code can access native APIs through the wrapper. Apache Cordova (<https://cordova.apache.org/>) is the most popular open source library to develop hybrid mobile apps.

Compared to native apps, hybrid apps have their benefits and drawbacks. The major benefit is that developers can use existing web development skills to create hybrid apps and use only one code base for different platforms. By leveraging responsive web design techniques, hybrid apps can easily adapt to different screen resolutions. The major drawback is the performance issues with hybrid apps. As the hybrid app is running inside of an internal browser, the performance of hybrid apps cannot compete with native apps. Certain types of apps, such as games or apps that rely on complicated native functionalities, cannot be built as hybrid apps. But many other apps can be built as hybrid apps.

Before making the decision of whether to go with native apps or hybrid apps, the development team needs to understand the nature of the apps to build. Hybrid apps are suitable for content-centric apps, such as news readers, online forums, or showcasing products. Another important factor to consider is the development team's skill sets. Most apps companies may need to hire both iOS and Android developers to support these two major platforms for native apps. But for hybrid apps, only front-end developers are enough. It's generally easier to hire front-end developers than Java or Swift/Objective-C developers.

Apache Cordova

Apache Cordova is a popular open source framework to develop hybrid mobile apps. It originates from PhoneGap (<http://phonegap.com/>) created by Nitobi. Adobe acquired Nitobi in 2011 and started to provide commercial services for it. The PhoneGap source code was contributed to the Apache Software Foundation and the new project Apache Cordova was started from its code base.

An Apache Cordova application is implemented as a web page. This web page can reference JavaScript, CSS, images, and other resources. The key component of understanding how Cordova works is the `WebView`. `WebView` is the component provided by native platforms to load and run web pages. Cordova applications run inside the `WebViews`. A powerful feature of Cordova is its plugin interface which allows JavaScript code running in a web page to communicate with native components. With the help of plugins, Cordova apps can access a device's accelerometer, camera, compass, contacts, and more. There are already many plugins available in Cordova's plugin registry (<http://cordova.apache.org/plugins/>).

Apache Cordova is just a runtime environment for web apps on native platforms. It can support any kind of web pages. To create mobile apps that look like native apps, we need other UI frameworks to develop hybrid mobile apps. Popular choices of hybrid mobile apps UI frameworks include Ionic framework (<http://ionicframework.com/>), Sencha Touch (<https://www.sencha.com/products/touch/>), Kendo UI (<http://www.telerik.com/kendo-ui>), and Framework7 (<http://framework7.io/>). Ionic framework is the one we are going to cover in this book.

Ionic Framework

Ionic framework is a powerful tool to build hybrid mobile apps. It's open source (<https://github.com/driftyco/ionic>) and has over 28,500 stars on GitHub, the popular social coding platform. Ionic framework is not the only player in hybrid mobile apps development, but it's the one that draws a lot

of attention and is recommended as the first choice by many developers. Ionic is popular for the following reasons:

- Use Angular (<https://angular.io/>) as the JavaScript framework. Since Angular is a popular JavaScript framework, the large number of Angular developers find it quite easy when moving to use Ionic for mobile apps development.
- Provide beautifully designed out-of-box UI components that work across different platforms. Common components include lists, cards, modals, menus, and pop-ups. These components are designed to have a similar look and feel as native apps. With these built-in components, developers can quickly create prototypes with good enough user interfaces and continue to improve them.
- Leverage Apache Cordova as the runtime to communicate with native platforms. Ionic apps can use all the Cordova plugins to interact with the native platform. Ionic Native further simplifies the use of Cordova plugins in Ionic apps.
- Performs great on mobile devices. The Ionic team devotes great effort to make it perform great on different platforms.

Ionic 2 (<http://ionic.io/2>) is a completely rewritten version of Ionic framework based on Angular 2. It dramatically improves performance and reduces the complexity of the code. It's recommended to use this new version of Ionic framework to build hybrid mobile apps. This book uses the 2.0.1 version of Ionic 2.

Firebase

Mobile apps usually need back-end services to work with the front-end UI. This means that there should be back-end code and servers to work with mobile apps. Firebase (<https://firebase.google.com/>) is a cloud service to power apps' back-end. Firebase can provide support for data storage and user authentication. After integrating mobile apps with Firebase, we don't need to write back-end code or manage the infrastructure.

Firebase works very well with Ionic to eliminate the pain of maintaining back-end code. This is especially helpful for hybrid mobile apps developers with only front-end development skills. Front-end developers can use JavaScript code to interact with Firebase.

Prepare Your Local Development Environment

Before we can build Ionic apps, we need to set up the local development environment first. We'll need various tools to develop, test, and debug Ionic apps.

Node.js

Node.js is the runtime platform for Ionic CLI. To use Ionic CLI, we need to install Node.js (<https://nodejs.org/>) on the local machine first. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It provides a way to run JavaScript on the desktop machines and servers. Ionic CLI itself is written in JavaScript and executed using Node.js. There are two types of release versions of Node.js – the stable LTS versions and current versions with the latest features. It's recommended to use Node.js version 6 or greater, especially the latest LTS version (6.9.4 at the time of writing).

Installing Node.js also installs the package management tool npm. npm is used to manage Node.js packages used in projects. Thousands of open source packages can be found in the npmjs registry (<https://www.npmjs.com/>). If you have a background with other programming languages, you may find npm is similar to Apache Maven (<https://maven.apache.org/>) for Java libraries or Bundler (<http://bundler.io/>) for Ruby gems.

Ionic CLI

After Node.js is installed, we can use npm to install Ionic command-line tools and Apache Cordova.

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

Note You may need to have system administrator privileges to install these two packages. For Linux and macOS, you can use `sudo`. For Windows, you can start a command-line window as the administrator. However, it's recommended to avoid using `sudo` when possible, as it may cause permission errors when installing native packages. Treat this as the last resort. The permission errors usually can be resolved by updating the file permissions of the Node.js installation directory.

After finishing installation of Ionic CLI and Cordova, we can use the command `ionic` to start developing Ionic apps.

You are free to use Windows, Linux, or macOS to develop Ionic 2 apps. Node.js is supported across different operating systems. One major limitation of Windows or Linux is that you cannot test iOS apps using the emulator or real devices. Some open source Node.js packages may not have the same test coverage on Windows as Linux or macOS. So they are more likely to have issues when running on Windows. But this should only affect the CLI or other tools, not Ionic 2 itself.

Yarn

Yarn (<https://yarnpkg.com/>) is a fast, reliable, and secure dependency management tool. After Facebook open sourced it, it quickly became popular in the Node.js community as a better alternative to npm. If you want to use yarn, follow the official instructions (<https://yarnpkg.com/en/docs/install>) to install it. After installing yarn, we can use the following command to install Ionic CLI and Cordova.

```
$ yarn global add cordova ionic
```

This book uses yarn instead of npm. If you didn't know about yarn before, read this guide (<https://yarnpkg.com/en/docs/migrating-from-npm>) about how to migrate from npm to yarn. Common yarn commands are listed below:

- `yarn add [package]` – Add packages as the project's dependencies. You can provide multiple packages to install. Version requirement can be specified following the Semantic Versioning spec (<http://semver.org/>).
- `yarn upgrade [package]` – Upgrade or downgrade versions of packages.
- `yarn remove [package]` – Remove packages.
- `yarn global` – Manage global dependencies.

The file `yarn.lock` contains the exact version of all resolved dependencies. This file is to make sure that builds are consistent across different machines. This file should be managed in the source code repository.

After Ionic CLI is installed, we can run `ionic info` to print out current runtime environment information and check for any warnings in the output; see Listing 1-1. The output also provides details information about how to fix those warnings.

Listing 1-1. Output of ionic info

Your system information:

```
Cordova CLI: 6.4.0
Ionic Framework Version: 2.0.1
Ionic CLI Version: 2.2.1
Ionic App Lib Version: 2.1.7
Ionic App Scripts Version: 1.0.0
ios-deploy version: 1.9.0
ios-sim version: 5.0.11
OS: macOS Sierra
Node Version: v6.9.4
Xcode version: Xcode 8.2.1 Build version 8C1002
```

iOS

Developing iOS apps with Ionic requires macOS and Xcode. You need to install Xcode and Xcode command-line tools on macOS. After installing Xcode, you can open a terminal window and type the command shown below.

```
$ xcode-select -p
```

If you see output like below, then command-line tools have already been installed.

```
/Applications/Xcode.app/Contents/Developer
```

Otherwise, you need to use the following command to install it.

```
$ xcode-select --install
```

After the installation is finished, you can use `xcode-select -p` to verify.

To run Ionic apps on the iOS simulator using Ionic CLI, package `ios-sim` is required. Another package `ios-deploy` is also required for deploying to install and debug apps. You can install both packages using the following command.

```
$ yarn global add ios-sim ios-deploy
```

Android

To develop Ionic apps for Android, Android SDK must be installed. Before installing Android SDK, you should have JDK installed first. Read this guide (<https://docs.oracle.com/javase/8/docs/technotes/guides/install/>) about how to install JDK 8 on different platforms. It's recommended to install

Android Studio (<https://developer.android.com/studio/index.html>) that provides a nice IDE and bundled Android SDK tools. If you don't want to use Android Studio, you can install stand-alone SDK tools.

Note Android API level 22 is required to run Ionic apps. Make sure that the required SDK platform is installed.

Stand-alone SDK tools is just a ZIP file; unpack this file into a directory and it's ready to use. The downloaded SDK only contains basic SDK tools without any Android platform or third-party libraries. You need to install the platform tools and at least one version of the Android platform. Run `android` in tools directory to start **Android SDK Manager** to install platform tools and other required libraries.

After installing Android SDK, you need to add SDK's tools and platform-tools directories into your PATH environment variable, so that SDK's commands can be found by Ionic. Suppose that the SDK tools is unpacked into `/Development/android-sdk`, then add `/Development/android-sdk/tools` and `/Development/android-sdk/platform-tools` to PATH environment variable. For Android Studio, the Android SDK is installed into directory `Users/<username>/Library/Android/sdk`.

To modify PATH environment variable on Linux and macOS, you can edit `~/.bash_profile` file to update PATH as shown below.

```
export PATH=${PATH}:/Development/android-sdk/platform-tools \
: /Development/android-sdk/tools
```

To modify PATH environment variable on Windows, you can follow the steps below.

1. Click **Start** menu, then right-click **Computer** and select **Properties**.
2. Click **Advanced System Settings** to open a dialog.
3. Click **Environment Variables** in the dialog and find **PATH** variable in the list, then click **Edit**.
4. Append the path of tools and platform-tools directories to the end of **PATH** variable.

It is highly recommended to use Android Studio instead of stand-alone SDK tools. Stand-alone SDK tools are more likely to have configuration issues.

Genymotion

Genymotion (<https://www.genymotion.com/>) is a fast Android emulator. It's recommended to use Genymotion for Android emulation instead of the standard emulators.

IDEs and Editors

You are free to use your favorite IDEs and editors when developing Ionic apps. IDEs and editors should have good support for editing HTML, TypeScript, and Sass files. For commercial IDEs, WebStorm (<https://www.jetbrains.com/webstorm/>) is recommended for its excellent support of various programming languages and tools. For open source alternatives, Visual Studio Code (<https://code.visualstudio.com/>) and Atom (<https://atom.io/>) are both popular choices.

Create an App Skeleton

After the local development environment is set up successfully, it's time to create new Ionic apps. Ionic 2 provides four different types of application templates. We can choose a proper template to create the skeleton code of the app. Apps are created using the command `ionic start`. The first argument of `ionic start` is the name of the new app, while the second argument is the template name. The `--v2` flag is also required when creating Ionic 2 apps; otherwise, Ionic 1 apps will be created instead.

The source code of these templates can be found on GitHub (<https://github.com/driftyco?query=ionic2-starter->). All these templates follow the same naming convention. The template names all start with `ionic2-starter-`. After removing the prefix `ionic2-starter-`, we can get the template name to be used in the command `ionic start`. For example, `ionic2-starter-blank` is the name of the template for blank apps.

Blank App

The template `blank` (<https://github.com/driftyco/ionic2-starter-blank>) only generates basic code for the app. This template should be used when you want to start from a clean code base; see Figure 1-1.

```
$ ionic start blankApp blank --v2
```

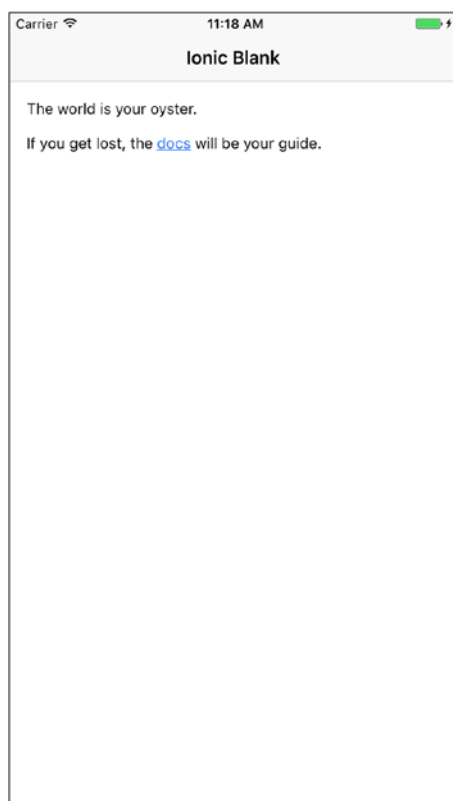


Figure 1-1. App created using the blank template

Tabbed App

The template tabs (<https://github.com/driftyco/ionic2-starter-tabs>) generates apps with a header at the top and tabs at the bottom; see Figure 1-2. This is also the default template when no template name is passed to the `ionic start`. This template should be used for apps with multiple views.

```
$ ionic start tabsApp tabs --v2
```

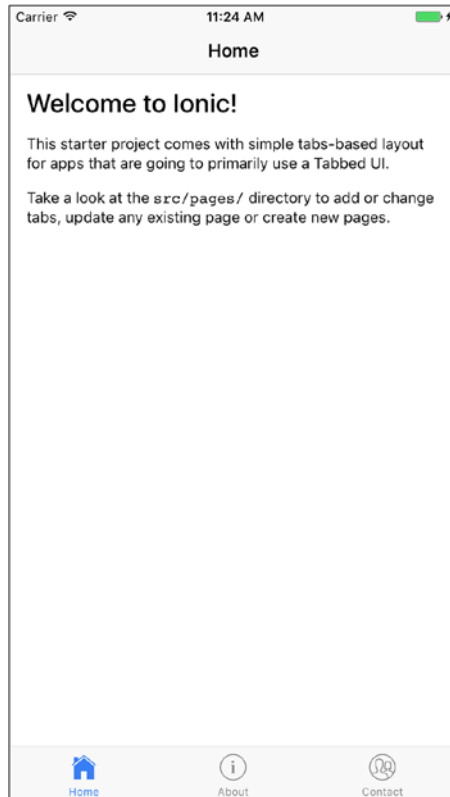


Figure 1-2. App created using the tabs template

Sidemenu

The template `sidemenu` (<https://github.com/driftyco/ionic2-starter-sidemenu>) generates apps with a side menu that opens itself when sliding to the left or clicking the menu icon; see Figure 1-3. This template can also be used for apps with multiple views, but it uses a menu to switch different views.

```
$ ionic start sidemenuApp sidemenu --v2
```

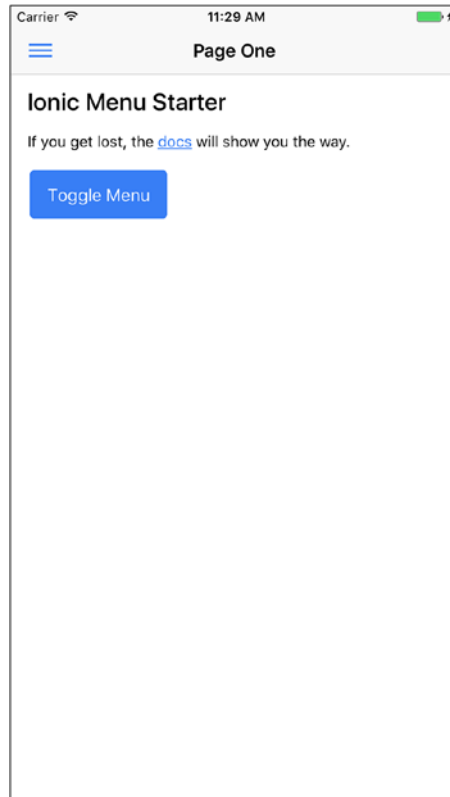


Figure 1-3. App created using the sidemenu template

Tutorial

The template tutorial (<https://github.com/driftyco/ionic2-starter-tutorial>) generates tutorial apps used with Ionic 2 official documentation; see Figure 1-4. This template is useful when you want to try live examples while reading through the official documentation. It's rarely used in actual development.

```
$ ionic start tutorialApp tutorial --v2
```

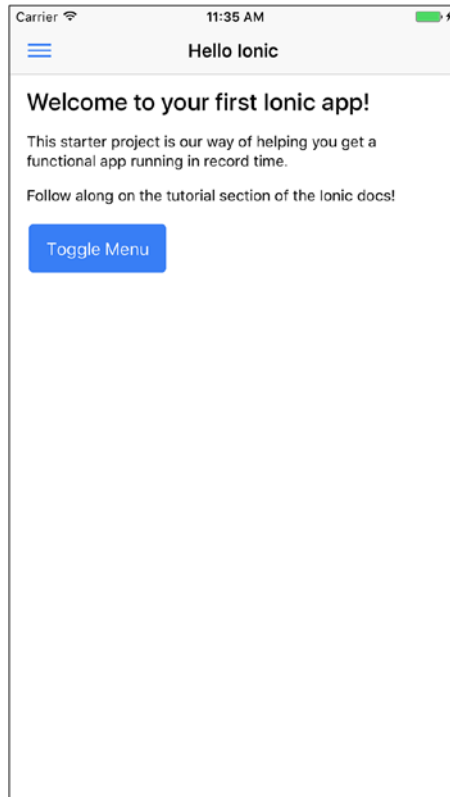


Figure 1-4. App created using the tutorial template

Local Development

After a new app is created using `ionic start`, we can navigate to the app directory and run `ionic serve` to start the local development server. The browser should automatically open a new window or tab that points to the address `http://localhost:8100/`. You should see the UI of this Ionic app. Ionic sets up livereload by default, so when any HTML, TypeScript, or Sass code is changed, it automatically refreshes the page to load the page with updated code. There is no need for a manual refresh.

The default port for the Ionic local development server is 8100. The port can be configured using `--port` or `-p` flag. For example, we can use `ionic serve -p 9090` to start the server on port 9090.

Use Chrome for Development

Using iOS or Android emulators to test and debug Ionic apps is not quite convenient because emulators usually consume a lot of system resources and take a long time to start or reload apps. A better alternative is to use Chrome browser for basic testing and debugging. To open Chrome DevTools, you can open Chrome system menu and select **More Tools > Developer Tools**. Once the developer tools window is opened, you need to click the mobile phone icon on the top menu bar to enable device mode. Then you can select different devices as rendering targets, for example, Apple iPhone 6 Plus or Nexus 6P. Figure 1-5 shows the app created from the tutorial template running in Chrome as Nexus 6P.

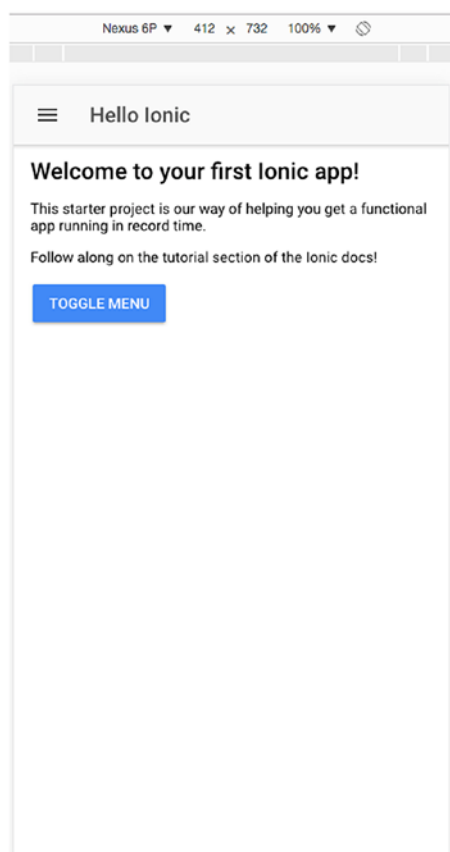


Figure 1-5. Use Chrome for development

Use Chrome DevTools for Android Remote Debugging

For Android platform, when an Ionic app is running on the emulator or a real device, we can use Chrome DevTools (<https://developers.google.com/web/tools/chrome-devtools/>) to debug the running app. Navigate to `chrome://inspect/#devices` in Chrome and you should see a list of running apps; see Figure 1-6. Clicking inspect launches the DevTools to inspect the running app. If you cannot see the app in the list, make sure that the device is listed in the result of the command `adb devices`.

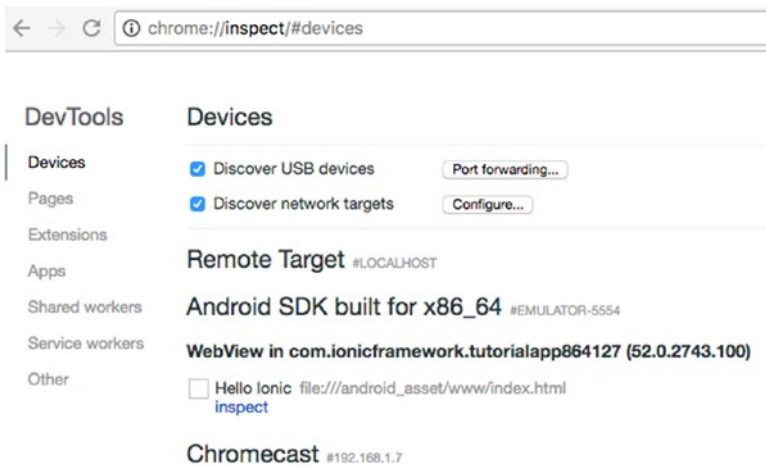


Figure 1-6. Chrome DevTools for Android remote debugging

Test on Emulators

After finishing the basic testing using browsers, it's time to test on device emulators. First, we need to configure the platforms support for the app.

iOS

We can use the following command to add iOS platform support. On macOS, iOS platform is added by default when creating the app using `ionic start`.

```
$ ionic platform add ios --save
```

Then the app can be built for iOS platform using the following command.

```
$ ionic build ios
```

Now you can start the emulator and test your app.

```
$ ionic emulate ios
```

Running the code above will launch the default iOS emulator. If you want to use a different emulator, you can use `--target` flag to specify the emulator name. To get a list of all the targets available in your local environment, use the following command.

```
$ cordova emulate ios --list
```

Then you can copy the target name from the output and use it in the `ionic emulate ios` command; see the code below to use the iPhone 6 Plus with iOS 10.1 emulator.

```
$ ionic emulate ios --target="iPhone-6-Plus, 10.1"
```

Android

To add Android platform, we can use the following command.

```
$ ionic platform add android --save
```

Then the app can be built for Android platform using the following command.

```
$ ionic build android
```

Now we can start the emulator and test the app; see Figure 1-7 for a screenshot of an Ionic app running on Android 7.1 emulator.

```
$ ionic emulate android
```

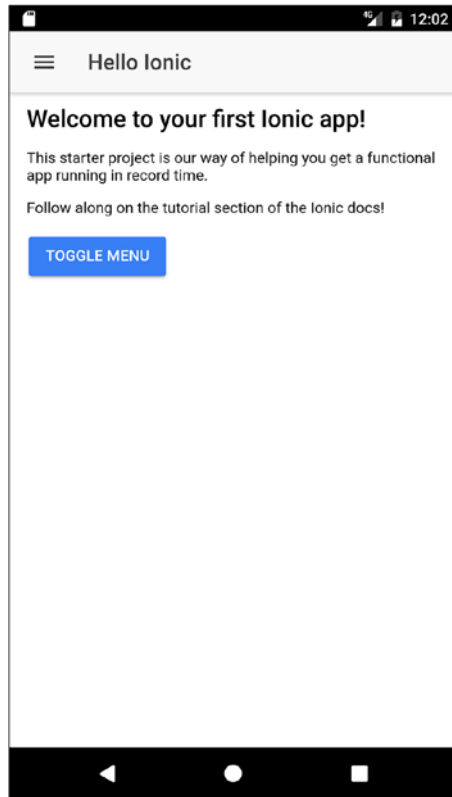


Figure 1-7. Ionic app running on Android 7.1 emulator

Note If you are using Genymotion for testing, you need to use the `ionic run android` command to run Ionic apps on the emulator; this is because Genymotion emulators are treated as real devices instead of emulators.

Summary

As the first chapter of this book, this chapter provides a basic introduction of hybrid mobile apps created with Ionic 2. After finishing this chapter, you should be able to set up your local environment to be ready for developing, debugging, and testing Ionic 2 apps. In the next chapter, we'll introduce programming languages, frameworks, libraries, and tools that are required in the actual development.

Chapter 2

Languages, Frameworks, Libraries, and Tools

Building hybrid mobile apps with Ionic 2 requires mostly front-end skills, including HTML, JavaScript, and CSS. You should have basic knowledge of these programming languages before reading this book. This chapter doesn't contain an introduction of these languages. You can find a lot of online resources if you do need to learn HTML, JavaScript, and CSS. Modern front-end development requires a lot more than just simple HTML, JavaScript, and CSS code. We need to deal with different languages, frameworks, libraries, and tools.

We'll discuss the following languages, frameworks, libraries, and tools in this chapter first.

- TypeScript
- Angular 2
- RxJS
- Sass
- Jasmine and Karma

This book only covers the basics of these languages, frameworks, libraries, and tools. You may need to read more related materials to understand their details. If you are confident that you have a good understanding of these, you can skip the whole chapter or related sections.

TypeScript

Other than standard JavaScript, Ionic 2 uses TypeScript (<https://www.typescriptlang.org/>) by default. This is because Angular 2 uses TypeScript by default. You are still free to use JavaScript if you don't want to learn a new programming language. But TypeScript is strongly recommended for enterprise applications development. As the name suggests, TypeScript adds type information to JavaScript. Developers with knowledge of other static-typing programming languages, for example, Java or C#, may find TypeScript very easy to understand. The official TypeScript documentation (<https://www.typescriptlang.org/docs/index.html>) is a good starting point to learn TypeScript.

Note It's no longer possible to use Ionic CLI to create Ionic 2 projects using JavaScript. Ionic CLI always uses TypeScript. Although it's still possible to bypass Ionic CLI and use JavaScript to build Ionic 2 apps, the configuration is quite complex and not recommended for most cases.

Why TypeScript?

The reason why TypeScript is recommended for Ionic apps development is because TypeScript offers several benefits compared to standard JavaScript.

Compile-Time Type Checks

TypeScript code needs to be compiled into JavaScript code before it can run inside of the browsers because browsers don't understand TypeScript. This process is called transpiling. During the compilation, the compiler does type checks using type declarations written in the source code. This static type check can eliminate potential errors in the early stage of development. JavaScript has no static-typing information in the source code. A variable declared with `var` can reference to any type of data. Even though this provides maximize flexibility when dealing with variables in JavaScript, it tends to cause more latent issues due to incompatible types in the runtime. For most of the variables and function arguments, their types are static and won't change in the runtime. For example, it's most likely an error will occur when assigning a string to a variable that should only contain a number. This kind of error can be reported by the TypeScript compiler in the compile time.

In Listing 2-1, the variable `port` represents the port that a server listens on. Even though this variable should only contain a number, it's still valid to assign a string `9090` to `port` in JavaScript. This error may only be detected in the runtime.

Listing 2-1. Valid JavaScript with type errors

```
var port = 8080;  
port = '9090';  
// -> valid assignment
```

However, the TypeScript code in Listing 2-2 declares the type of port is number. The following assignment causes a compiler error. So developers can find out this error immediately and fix it right away.

Listing 2-2. Type checking in TypeScript

```
let port: number = 8080;  
port = '9090';  
// -> compiler error!
```

Rich Feature Sets

Apart from the essential compile-time type checks, TypeScript is also a powerful programming language with rich feature sets. Most of these features come from current or future versions of ECMAScript, including ES6 and ES7. Using these features can dramatically increase the productivity of front-end developers. You'll see the usages of these features in the code of the sample app.

Better IDE Support

With type information in the TypeScript source code, modern IDEs can provide smart code complete suggestions to increase developers' productivity. IDEs can also do refactoring for TypeScript code. Navigation between different files, classes, or functions is easy and intuitive. Front-end developers can enjoy the same coding experiences as Java and C# developers.

Basic Types

The key point of writing TypeScript code is to declare types for variables, properties, and functions. TypeScript has a predefined set of basic types. Some of those types come from JavaScript, while other types are unique in TypeScript.

Boolean

Boolean type represents a simple true or false value. A Boolean value is declared using type boolean in TypeScript.

```
let isActive: boolean = false;  
isActive = true;
```

Number

Numbers are all floating-point values in TypeScript. A number is declared using type `number` in TypeScript. TypeScript supports decimal, hexadecimal, binary, and octal literals for numbers. All these four numbers in the code below have the same decimal value 20.

```
let n1: number = 20;           // decimal
let n2: number = 0x14;         // hexadecimal
let n3: number = 0b10100;      // binary
let n4: number = 0o24;         // octal
```

String

String type represents a textual value. A string is declared using type `string` in TypeScript. Strings are surrounded by double quotes (") or single quotes ('). It's up to the development team to choose whether to use double quotes or single quotes. The key point is to remain consistent across the whole code base. Single quotes are more popular because they are easier to type than double quotes that require the shift key.

```
let text: string = 'Hello World';
```

TypeScript also supports ES6 template literals, which allow embedded expressions in string literals. Template literals are surrounded by backticks (```). Expressions in the template literals are specified in the form of `${expression}`.

```
let a: number = 1;
let b: number = 2;
let result: string = `${a} + ${b} = ${a + b}`;
// -> string "1 + 2 = 3"
```

Null and Undefined

`null` and `undefined` are special values in JavaScript. In TypeScript, `null` and `undefined` also have a type with name `null` and `undefined`, respectively. These two types only have a single value.

```
let v1: null = null;
let v2: undefined = undefined;
```

By default, it's allowed to assign `null` and `undefined` to variables declared with other types. For example, the code below assigns `null` to the variable `v` with type `string`.

```
let v: string = null;
```

However, null values generally cause errors in the runtime and should be avoided when possible. TypeScript compiler supports the option `-strictNullChecks`. When this option is enabled, TypeScript compiler does a strict check on null and undefined values. null and undefined can only be assigned to themselves and variables with type any. The code above will have a compile error when `strictNullChecks` is enabled.

Array

Array type represents a sequence of values. The type of an array depends on the type of its elements. Appending `[]` to the element type creates the array type. In the code below, `number[]` is the type of arrays with numbers, while `string[]` is the type of arrays with strings. Array type can also be used for custom classes or interfaces. For example, `Point[]` represents an array of Point objects.

```
let numbers: number[] = [1, 2, 3];
let strings: string[] = ['a', 'b', 'c'];
```

Tuple

The elements of an array are generally of the same type, that is, a homogeneous array. If an array contains a fixed number of elements of different types, that is, a heterogeneous array, it's called a tuple. The tuple type is declared as an array of element types. In the code below, the tuple `points` has three elements with types `number`, `number`, and `string`.

```
let points: [number, number, string] = [10, 10, 'P1'];
```

Tuples are useful when returning multiple values from a function because a function can only have at most one return value. Tuples of two elements, a.k.a. pairs are commonly used. Be careful when using tuples with more than two elements, because elements of tuples can only be accessed using array indices, so it reduces the code readability. In this case, tuples should be replaced with objects with named properties. So it's better to change the type of `points` to an actual interface.

Enum

Enum type represents a fixed set of values. Each value in the set has a meaningful name and a numeric value associated with the name. In the code below, the value of `status` is a number with value 1. By default, the numeric

values of enum members start from 0 and increase in sequence. In the enum `Status`, `Status.Started` has value 0, `Status.Stopped` has value 1, and so on.

```
enum Status { Started, Stopped, Error };  
let status: Status = Status.Stopped;
```

It's also possible to assign specific numeric values to enum values. In the code below, enum values `Read`, `Write`, and `Execute` have their assigned values. The value of `permission` is 3.

```
enum Permission { Read = 1, Write = 2, Execute = 4 };  
let permission = Permission.Read | Permission.Write;
```

To convert an enum value back to its textual format, we can do the lookup by treating the enum type as an array.

```
let status: string = Status[1];  
// -> 'Stopped'
```

Any

Any type is the escape bridge from the TypeScript world to the JavaScript world. When a value is declared with type `any`, no type checking is done for this value. While type information is valuable, there are some cases when type information is not available, so we need the `any` type to bypass the compile-time check.

- Migrate a JavaScript code base to TypeScript. During the migration, we can annotate unfinished values as `any` to make the TypeScript code compile.
- Integrate with third-party JavaScript libraries. If TypeScript code uses a third-party JavaScript library, we can declare values from this library as `any` to bypass type checks for this library. However, it's better to add type definitions for this kind of libraries, either by loading type definitions from community-driven repositories or creating your own type definitions files.

In the code below, the variable `val` is declared as `any` type. We can assign a `string`, a `number`, and a `Boolean` value to it.

```
let val: any = 'Hello World';  
val = 100; // valid  
val = true; // valid
```

Void

Void means no type. It's commonly used as the return type of functions that don't return a value. The return type of the `sayHello` function below is `void`.

```
function sayHello(): void {  
    console.log('Hello');  
}
```

`void` can also be used as a variable type. In this case, the only allowed values for this variable are `undefined` and `null`.

Never

`Never` is a special type related to control flow. It represents the type of values that should never occur. If a function always throws an exception or it contains an infinite loop that makes the function never return, the return type of this function is `never`. The function `neverHappens` in the code below always throws an `Error` object, so its return type is `never`.

```
function neverHappens(): never {  
    throw new Error('Boom!');  
}
```

Union

Union type represents a value that can be one of several types. The allowed types are separated with a vertical bar (`|`). In the code below, the type of the variable `stringOrNumber` can be either `string` or `number`.

```
let stringOrNumber: string | number = 'Hello World';  
stringOrNumber = 3;  
stringOrNumber = 'Test';
```

Union types can also be used to create enum-like string literals. In the code below, the type `TrafficColor` only allows three values.

```
type TrafficColor = 'Red' | 'Green' | 'Yellow';  
let color: TrafficColor = 'Red';
```

Functions

Functions are important building blocks of JavaScript applications. TypeScript adds type information to functions. The type of a function is defined by the types of its arguments and return value.

As shown in Listing 2-3, we only need to declare function types either on the variable declaration side or on the function declaration side. TypeScript compiler can infer the types from context information.

Listing 2-3. Declaring function types

```
let size: (str: string) => number = function(str) {  
    return str.length;  
};  
  
let multiply = function(v1: number, v2: number): number {  
    return v1 * v2;  
}
```

Function types are useful when declaring high-order functions, that is, functions that take other functions as arguments or return other functions as results. When specifying types of functions used as arguments or return values, only type information is required, for example, `(string) => number` or `(number, number) => number`. We don't need to provide the formal parameters. In Listing 2-4, `forEach` is a high-order function that takes functions of type `(any) => void` as the second argument.

Listing 2-4. Use function types in high-order functions

```
function forEach(array: any[], iterator: (any) => void) {  
    for (let item in array) {  
        iterator(item);  
    }  
}  
  
forEach([1, 2, 3], item => console.log(item));  
// -> Output 1, 2, 3
```

Arguments

JavaScript uses a very flexible strategy to handle function arguments. A function can declare any number of formal parameters. When the function is invoked, the caller can pass any number of actual arguments. Formal parameters are assigned based on their position in the arguments list. Extra arguments are ignored during the assignment. When not enough number of arguments are passed, missing formal parameters are assigned to `undefined`. In the function body, all the arguments can be accessed using the array-like `arguments` object: for example, using `arguments[0]` to access the first actual argument. This flexibility of arguments handling is a powerful feature and enables many elegant solutions with arguments manipulation in JavaScript. However, this flexibility causes unnecessary burden on developers to understand. TypeScript adopts a stricter restriction

on arguments. The number of arguments passed to a function must match the number of formal parameters declared by this function. Passing more or fewer arguments when invoking a function is a compile-time error.

If a parameter is optional, we can add `?` to the end of the parameter name, then the compiler doesn't complain when this parameter is not provided when invoking this function. Optional parameters must come after all the required parameters in the function's formal parameters list. Otherwise, there is no way to correctly assign arguments to those parameters. For example, given a function `func(v1?: any, v2: any, v3?: any)`, when it's invoked using `func(1, 2)`, we could not determine whether value 1 should be assigned to `v1` or `v2`.

We can also set a default value to a parameter. If the caller doesn't provide a value or the value is undefined, the parameter will use the default value. In Listing 2-5, the parameter `timeout` of function `delay` has a default value 1000. The first invocation of `delay` function uses the default value of `timeout`, while the second invocation uses the provided value 3000.

Listing 2-5. Default value of a parameter

```
function delay(func: () => void, timeout = 1000) {
    setTimeout(func, timeout);
}

delay(() => console.log('Hello'));
// -> delay 1000ms
delay(() => console.log('Hello'), 3000);
// -> delay 3000ms
```

TypeScript also supports using ellipses (`...`) to collect all remaining unmatched arguments into an array. This allows developers to deal with a list of arguments. In Listing 2-6, all the arguments of the `format` function are collected into the array `values`.

Listing 2-6. Collect unmatched arguments

```
function format(...values: string[]): string {
    return values.join('\n');
}

console.log(format('a', 'b', 'c'));
```

Interfaces and Classes

TypeScript adds common concepts from object-oriented programming languages. This makes it very easy for developers familiar with other object-oriented programming languages move to TypeScript.

Interfaces

Interfaces in TypeScript have two types of usage scenarios. Interfaces can be used to describe the shape of values or act as classes contracts.

Describe the shape of values

In typical JavaScript code, we use plain JavaScript objects as the payload of communication. But the format of these JavaScript objects is opaque. The caller and receiver need to implicitly agree on the format, which usually involves collaboration between different developers. This type of opacity usually causes maintenance problems.

For example, a receiver function may accept an object that contains the properties `name`, `email`, and `age`. After a later refactoring, the development team found that `date of birth` should be passed instead of the `age`. The caller code was changed to pass an object that contains the properties `name`, `email`, and `dateOfBirth`. Then the receiver code failed to work anymore. This kind of errors can only be found in the runtime if developers failed to spot all those places that rely on this hidden data format contract during refactoring. Because of this potential code breaking, developers tend to only add new properties while keeping those old properties, even though those properties were not used anymore. This is bad for the quality of the whole code base.

Interfaces in TypeScript provide a way to describe the shape of an object. As shown in Listing 2-7, if we update interface `User` to remove the property `age` and add a new property `dateOfBirth`, TypeScript compiler will throw errors on all the places where the property `age` is used in the whole code base. This is a huge benefit for code refactoring and maintenance.

Listing 2-7. Use interface to describe the shape of values

```
interface User {
  name: string;
  email: string;
  age: number;
}

function processUser(user: User) {
  console.log(user.name);
}

processUser({
  name: 'Alex',
  email: 'alex@example.org',
  age: 34,
});
```

interface can also be used to describe function types, which can provide consistent names for function types. Considering the `forEach` function in Listing 2-8, interface `Iterator` describes the function type of the second parameter.

Listing 2-8. Use interface to describe function types

```
interface Iterator {
    (item: any): void
}

function forEach(array: any[], iterator: Iterator) {
    for (let item in array) {
        iterator(item);
    }
}

forEach([1, 2, 3], item => console.log(item));
// -> Output 1, 2, 3
```

As classes contracts

Interfaces in TypeScript can also be used as contracts for classes. This is the typical usage of interfaces in object-oriented programming languages like Java or C#. In Listing 2-9, interface `DataLoader` has two implementations, `DatabaseLoader` and `NetworkLoader`. These two classes have different implementations of the method `load()`.

Listing 2-9. Interfaces as classes contracts

```
interface DataLoader {
    load(): string;
}

class DatabaseLoader implements DataLoader {
    load() {
        return 'DB';
    }
}

class NetworkLoader implements DataLoader {
    load() {
        return 'Network';
    }
}

let dataLoader: DataLoader = new DatabaseLoader();
console.log(dataLoader.load());
// -> Output 'DB'
```

Classes

Class is the fundamental concept in object-oriented programming languages. ES6 added the classes concept to JavaScript. TypeScript also supports classes.

Listing 2-10 shows important aspects of classes in TypeScript. A class can be abstract. An abstract class cannot be instantiated directly, and it contains abstract methods that must be implemented in derived classes. Classes also support inheritance. The members of a class are public by default. `public`, `protected`, and `private` modifiers are supported with similar meanings as in other object-oriented programming languages.

Classes can have constructor functions to create new instances. In the constructor function of a subclass, the constructor of its parent class must be invoked using `super()`. The constructor of `Rectangle` takes two parameters `width` and `height`, but the constructor of `Square` takes only one parameter, so `super(width, width)` is used to pass the same value `width` for both parameters in `Rectangle` constructor function.

Listing 2-10. Classes

```
abstract class Shape {
    abstract area(): number;
}

class Rectangle extends Shape {
    private width: number;
    private height: number;
    constructor(width: number, height: number) {
        super();
        this.width = width;
        this.height = height;
    }
    area() {
        return this.width * this.height;
    }
}

class Square extends Rectangle {
    constructor(width: number) {
        super(width, width);
    }
}

class Circle extends Shape {
    private radius: number;
    constructor(radius: number) {
        super();
    }
}
```

```
    this.radius = radius;
  }
  area() {
    return Math.PI * this.radius * this.radius;
  }
}
```

```
let rectangle = new Rectangle(5, 4);
let square = new Square(10);
let circle = new Circle(10);
```

```
console.log(rectangle.area());
// -> 20
console.log(square.area());
// -> 100
console.log(circle.area());
// -> 314.1592653589793
```

Decorators

Decorator is an experimental yet powerful feature in JavaScript. It's the JavaScript way of adding annotations and meta-programming support. Java developers may find decorators are easy to understand since Java already has annotations with similar syntax. Since the support of decorators is now experimental, it needs to be enabled explicitly using `experimentalDecorators` option on TypeScript compiler. Understanding decorators is important as they are fundamental building blocks in Angular 2.

A decorator can be attached to a class declaration, method, accessor, property, or parameter. Decorators are declared using the form `@expression`, where `expression` must be evaluated to a function that will be invoked with information about the decorated element. In the decorator's function, it can retrieve, modify, and replace the definition it decorates. You can add multiple decorators to the same class declaration, method, property, or parameter.

In practice, it's common to use another function to create the actual decorator functions. This kind of functions is called `decorator factories`. Decorator factories allow further customizations on how decorators are applied in the runtime.

Class Decorators

A class decorator is added to a class declaration and applied to the constructor of the class. The decorator function receives the constructor as the only argument. If the decorator function returns a value, the value is used as the new constructor that replaces the existing constructor. Class

decorators allow us to replace the classes constructors when necessary. The decorator function is called when the class is declared. Instantiation of objects of a class doesn't call the class decorators on it.

In Listing 2-11, we create a class decorator `refCount` to keep track of the number of instances created for a class. `refCount` is a decorator factory that takes a name argument that is used as the key in the data structure `refCountMap`. The decorator function creates a new constructor `newCtor` that records the count and invokes the original constructor. The new constructor replaces existing constructor of the decorated class.

Listing 2-11. Class decorators

```
@refCount('sample')
class Sample {
  constructor(value: string) {

  }
}

const refCountMap = {};

function refCount(name: string) {
  return (constructor: any) => {
    const originalCtor = constructor;
    const newCtor:any = function (...args) {
      if (!refCountMap[name]) {
        refCountMap[name] = 1;
      } else {
        refCountMap[name]++;
      }
      return originalCtor.apply(this, args);
    }
    newCtor.prototype = originalCtor.prototype;
    return newCtor;
  };
}

const s1 = new Sample('hello');
const s2 = new Sample('world');
console.log(refCountMap['sample']);
// -> 2
```

Method Decorators

A method decorator is added to a method declaration. The decorator function receives three arguments:

1. `target` – For a static method of a class, it's the constructor function; for an instance method, it's the prototype of the class.
2. `propertyKey` – The name of the function.
3. `descriptor` – The `PropertyDescriptor` that contains information about the function.

In Listing 2-12, we create a method decorator `fixedValue` that makes a function return provided a fixed value regardless of the actual argument. In the decorator function, we set the value of the method's `PropertyDescriptor` instance to a function that simply returns the provided value. Updating the value property of the `PropertyDescriptor` instance replaces the method declaration.

Listing 2-12. Method decorator

```
class Sample {
  @fixedValue('world')
  test() {
    return 'hello';
  };
}

function fixedValue(value: any) {
  return (target: any, propertyKey: string, descriptor: PropertyDescriptor)
    => {
    descriptor.value = () => value;
  };
}

const sample = new Sample();
console.log(sample.test());
// -> 'world'
```

The `PropertyDescriptor` contains other properties like `writable`, `enumerable`, and `configurable`. These properties can also be updated.

Property Decorators

A property decorator is added to a property declaration. The decorator function receives two arguments: `target` and `propertyKey`, which are the same as the first two arguments of method decorators. To replace a property in the decorator function, we need to return a new `PropertyDescriptor` object.

In Listing 2-13, we create a property decorator `fixedValue` with the same functionality as the method decorator in Listing 2-12 but that applies to properties. This property decorator `fixedValue` sets the property's value to the provided value. The return value of the decorator function is the property descriptor of the new property. Here we only set the property value of the `PropertyDescriptor` and use default values for other properties.

Listing 2-13. Property decorator

```
class Sample {
  @fixedValue('world')
  value: string = 'hello';
}

function fixedValue(value: any) {
  return (target: any, propertyKey: string) => {
    return {
      value,
    };
  };
}

const sample = new Sample();
console.log(sample.value);
// -> 'world'
```

Angular 2

Angular 2 is the next version of the popular JavaScript framework Angular 1. Angular 2 is a full-fledged framework for developing web applications. Angular 2 itself is out of the scope of this book. There are plenty of online resources about learning Angular 2. Ionic 2 apps are built using Angular 2, so it's important for developers to have a basic understanding of Angular 2 when building Ionic 2 apps. This section covers essential parts of Angular 2. More detailed documentation can be found on the Angular 2 official site (<https://angular.io/docs/ts/latest/>).

We'll go through following core building blocks of Angular 2.

- Modules
- Components
- Templates
- Metadata
- Data binding

- Directives
- Services
- Dependency injection

Modules

Angular modules provide the solutions to break application logic into separate units. Each app must have at least one module, the root module, for bootstrapping the whole app. For small apps, one module is generally enough. For complex apps, it's common to have different modules grouping by functionalities.

From the implementation point of view, a module is just a class with a `NgModule` decorator factory. When declaring a module, we just need to provide necessary metadata to the `NgModule()` method and Angular will create the module instance. Metadata of the module is provided as properties of the only parameter of `NgModule()`. Here we list important properties of `NgModule` metadata.

- `declarations` – Components, directives, and pipes that belong to this module.
- `exports` – Set of declarations that are usable in other modules.
- `imports` – Set of imported modules. Declarations exported from imported modules are usable in the current module.
- `providers` – Set of services that are available for dependency injection in this module.
- `entryComponents` – Set of components that should be compiled when the module is defined.
- `bootstrap` – Set of components that should be bootstrapped when the module is bootstrapped. This property should only be set for the root module.

Listing 2-14 declares a module `MyModule`. It declares a component `MyComponent`, a service `MyService`, and imports `CommonModule`.

Listing 2-14. Angular 2 module

```
@NgModule({  
  declarations: [  
    MyComponent,  
  ],  
})
```

```
providers: [  
  MyService,  
],  
imports: [  
  CommonModule,  
]  
})  
export class MyModule {  
  
}
```

Components

A component is responsible for managing a portion of an app's user interface. An Angular 2 app consists of a hierarchical structure of components. A component has a template associated with it. The template defines the HTML markup to display when the component is rendered.

The rendered HTML of a component usually depends on the data passed to the component. Some parts of a template may use a special syntax to define the binding to the data passed to it. During the rendering, those placeholders in the template will be replaced with data values come from runtime, for example, from server-side or local storage. Data binding makes writing dynamic HTML elegant and easy to maintain.

Directives are used to attach extra behaviors to standard DOM elements. Structural directives can add, remove, or replace DOM elements. For example, `ngFor` creates repeated elements based on an array of items. `ngIf` only adds elements when the predicate matches. Attribute directives change the appearance or behavior of an element. For example, `ngClass` adds extra CSS classes to the element.

A component is created by annotating a class with a `Component` decorator factory. `Component` decorator supports a list of different properties to configure the component, including the following:

- `template` – Inline HTML template for the view.
- `templateUrl` – URL to an external template file for the view.
- `selector` – CSS selector for this component when referenced in other templates.
- `style` – Inline CSS styles for the view.
- `styleUrls` – List of URLs to external stylesheets.
- `providers` – List of providers available for the component and its children.

Listing 2-15 creates a component `MyComponent` with the selector `my-component`. It can be created using `<my-component></my-component>` in other components.

Listing 2-15. Angular 2 component

```
@Component({
  template: '<p>Hello {{name}}!</p>',
  selector: 'my-component',
})
export class MyComponent {
  name: string = 'Alex';
}
```

Template syntax

In Angular 2, we can embed different types of expressions in the template for data binding.

- `{{expr}}` – String interpolation to be used in HTML text or attribute assignments.
- `[name]` – Property binding, for example, ``.
- `(event)` – Event binding, for example, `<button (click)="save()">Save</button>`.
- `[(ngModel)]` – Two-way property and event binding, for example, `<input [(ngModel)]="username">`.
- `[attr.attr-name]` – Attribute binding, for example, `<td [attr.colspan]="colspan"></td>`.
- `[class.class-name]` – class property binding, for example, `<div [class.header]="isHeader"></div>`.
- `[style.style-property]` – style property binding, for example, ``.

Services

Unlike modules or components, Angular 2 doesn't have a definition for services. There is no limitation on what could be a service. It can be a class or a value. Services are designed to encapsulate the logic that is unrelated to the view. Components should only handle view presentation and user interactions. All other logic, like communication with the back-end server, logging, and input validation, should be organized into services.

Listing 2-16 shows a `Logger` service that logs different types of messages. The `Injectable` decorator factory makes the `Logger` service available in dependency injection.

Listing 2-16. Angular 2 service

```
@Injectable()
export class Logger {
  debug(message: string) {}
  info(message: string) {}
  warn(message: string) {}
  error(message: string) {}
}
```

Dependency Injection

Services can be used everywhere in a module. Components can declare their dependencies to services using constructor parameters. `Injector` in Angular is responsible for providing the actual instances of services when components are instantiated.

The injector creates instances of services when necessary. It needs to know how to create new instances. This is done by modules or components to declare providers. There are three types of providers.

Class providers

This is the default type of providers. Class providers are specified using the classes. New instances are created by invoking the constructors. For example, `providers: [Logger]` in the module declaration means using the constructor of `Logger` class to create new instances. This is actually a shorthand expression of `[{ provide: Logger, useClass: Logger }]`. The property `provide` specifies the token to get the created instance. We can change to a different class using the property `useClass`, e.g. `[{ provide: Logger, useClass: AnotherLogger }]`.

Value providers

Value providers use existing objects as the values for the injector. For example, `[provide: Logger, useValue: myLogger]` uses the object `myLogger` as the value of `Logger` instances.

Factory providers

Sometimes, we need to use complex logic to create new instances, or external dependencies are required for the creation. In this case, we can use the factory providers that are custom functions. The parameters of provider functions can be dependencies resolved by the injector.

In Listing 2-17, `loggerFactory` is the function to create new instances of `Logger` service. It accepts two arguments of types `StorageService` and `UserService`.

Listing 2-17. Function to create new service instances

```
const loggerFactory = (storageService: StorageService, userService:
UserService) => {
  return new Logger();
}
```

Listing 2-18 shows how to use factory provider. `loggerServiceProvider` is the provider for `Logger` service. The property `useFactory` specifies the provider function `loggerFactory`. The property `deps` specifies the dependent services as the arguments of `loggerFactory`.

Listing 2-18. Use factory provider

```
const loggerServiceProvider =
{ provide: Logger,
  useFactory: loggerFactory,
  deps: [StorageService, UserService],
};
```

Tokens

In the previous examples of using providers, we always use the class type as the value of the property `provide`. The value is called dependency injection token, which is the key to get the associated provider. If we don't have a class for the provider, we can use an `OpaqueToken`. In Listing 2-19, we create an `OpaqueToken` `TITLE` with value `app.title`.

Listing 2-19. Dependency injection tokens

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

export let TITLE = new OpaqueToken('app.title');
```

Then we can use this `OpaqueToken` in the property `provide`.

```
providers: [{ provide: TITLE, useValue: 'My app' }]
```


Because the `OpaqueToken` has no class type, we need to use the `Inject` decorator to inject it into constructors.

```
constructor(@Inject(TITLE) title: string) {  
}
```

Use services in components

To use the services in a component, we need to add the service to the array of providers in the `NgModule` decorator, as shown in [Listing 2-20](#).

Listing 2-20. Add services to module

```
@NgModule({  
  providers: [  
    logger,  
  ]  
})  
export class MyModule {  
  
}
```

We only need to add the service dependency in the constructor. The Angular 2 injector injects the instance when the component is created, as shown in [Listing 2-21](#).

Listing 2-21. Add service dependency in the constructor

```
export class MyComponent {  
  constructor(public logger: Logger) {  
  }  
}
```

Metadata

Metadata is related to how Angular 2 works internally. Angular 2 uses decorators to declaratively add behaviors to normal classes. Metadata is provided when using decorators. We already see the usage of metadata in `NgModule` and `Component` decorators. Those properties passed to the decorator factories are metadata.

Bootstrapping

An Angular app is started by bootstrapping the root module. Angular apps can run on different platforms, for example, browsers or server-side. In [Listing 2-22](#), we use the browser dynamic platform to bootstrap the `AppModule`.

Listing 2-22. Bootstrap Angular 2 app

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

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

RxJS

RxJS is another complicated yet important library used in Angular 2 apps. RxJS is very powerful for apps using asynchronous and event-based data processing. The core of RxJS is observable sequences that allow pushing of multiple values into consumers synchronously or asynchronously. A RxJS observable can produce multiple values at its own pace. An observer that is interested in the observable can subscribe to the observable by providing callback functions for different scenarios. These callback functions will be invoked when the observable produce different types of values.

This section only provides a basic introduction of RxJS; see this link (<http://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/>) for in-depth discussion.

Observable

RxJS provides two different approaches to create observables. The first approach is using `Observable.create()` method. The argument of the `create()` method is a function that defines how values are produced. The function receives an observer instance and uses methods `next()`, `complete()` and `error()` to produce different notifications.

- `next` – Produce a normal value.
- `complete` – Produce a notification so that the observable completes its value production.
- `error` Produce a notification that an error occurred during its value production.

`complete` and `error` notifications are terminal, which means there won't be any values for the observable after these two types of notifications.

In Listing 2-23, we create an observable that produces two values `hello` and `world`, then produces the `complete` notification.

Listing 2-23. Create Observables with observers

```
Observable.create(observer => {  
  observer.next('hello');  
  observer.next('world');  
  observer.complete();  
});
```

The second approach is using built-in RxJS operators to create observables. In Listing 2-24, `Observable.of()` method creates an observable with two values. `Observable.from()` method creates an observable from an iterable object.

Listing 2-24. Create Observables with operators

```
Observable.of('hello', 'world');  
// -> Observable with values "hello" and "world"  
  
Observable.from([1, 2, 3]);  
// -> Observable with values 1, 2 and 3
```

Observers

To receive notifications from an observable, we need to subscribe to it using the `subscribe()` method. When subscribing to an observable, we can provide any number of callback handlers for those three different types of notifications. In the code below, we only add callback handler for the next notification and ignore the error and complete notifications. In the handler of next notification, we simply log out the value to the console.

```
Observable.from([1, 2, 3]).subscribe(v => console.log(v));
```

The return value of `subscribe()` method is an instance of RxJS Subscription class that can be used to manage the subscription. It's important to call the `unsubscribe()` method to release resources or cancel executions when the subscription is no longer used.

```
const subscription = Observable.from([1, 2, 3]).subscribe(v =>  
  console.log(v));  
subscription.unsubscribe();
```

Subjects

A subject is a special type of Observable that can be both an Observable and an Observer. As an Observable, we can subscribe to it to receive notifications. As an Observer, we can use `next(v)`, `error(e)` and `complete()` methods to

send different notifications. These notifications will be multicasted to other observers that subscribed to the subject, as shown in Listing 2-25.

Listing 2-25. Subjects

```
const subject = new Subject();
subject.subscribe(v => console.log(v));

subject.next(1);
subject.next(2);
subject.next(3);
```

Operators

The power of RxJS comes from all the provided operators. When an operator is invoked on an `Observable`, it returns a new `Observable`. The previous `Observable` stays unmodified.

In Listing 2-26, for the given `Observable`, we use a `map` operator to change the values to be multiplied by 100, then we use a `filter` operator to filter the values using given predicate function. The final result `Observable` contains only value 300.

Listing 2-26. Operators

```
Observable.from([1, 2, 3])
  .map(v => v * 100)
  .filter(v => v % 3 == 0)
  .subscribe(v => console.log(v));
```

Sass

Sass is a powerful CSS extension language. It belongs to the group of CSS preprocessors with other alternatives like LESS and Stylus. These CSS preprocessors take code written in their own format and turn it into CSS stylesheets. Preprocessors allow developers to use latest features in CSS, which is the same motivation of using TypeScript over JavaScript.

Variables

We can define variables in Sass files, just like other programming languages. Variables are defined using `$` as the prefix. See Listing 2-27.

Listing 2-27. Sass variables

```
$primary-color: red;

h1 {
  color: $primary-color;
}
```

Nesting

We can nest CSS selectors in Sass to match the visual hierarchy of HTML elements. See Listing 2-28.

Listing 2-28. Sass nesting

```
article {
  padding: 10px;

  header {
    font-weight: bold;
  }

  section {
    font-size: 12px;
  }
}
article {
  padding: 10px;
}
article header {
  font-weight: bold;
}
article section {
  font-size: 12px;
}
```

Mixins

Mixins, as shown in Listing 2-29, are reusable functions that can be shared in different modules.

Listing 2-29. Sass mixins

```
@mixin important-text($font-size) {
  font-weight: bold;
  font-size: $font-size;
}
```

```
header {
  @include important-text(12px);
}
header {
  font-weight: bold;
  font-size: 12px;
}
```

Jasmine and Karma

Jasmine (<https://jasmine.github.io/>) is a behavior-driven development framework for testing JavaScript code. Testing code with Jasmine is organized as suites. Each suite can include multiple specs. A spec contains one or more expectations that represent the actual testing logic to run.

A suite is created by calling the `describe()` method with a string as the name and a function as the body to execute. A spec is created by calling the `it()` method with the same parameters as `describe()` method. An expectation is created by calling the `expect()` method with the value to check, then chained with different matcher functions for verifications.

Listing 2-30 is a simple suite with one spec.

Listing 2-30. Jasmine test suite

```
describe('A sample suite', () => {
  it('true is true', () => {
    expect(true).toBe(true);
  });
});
```

In the suite, we can also use `beforeEach()`, `afterEach()`, `beforeAll()` and `afterAll()` methods to add custom setup and teardown code.

Karma (<https://karma-runner.github.io/>) is the tool to run tests on different platforms. We use Karma to run tests written with Jasmine. Karma can launch different browsers to run the tests.

Summary

Building Ionic 2 apps requires knowledge of different programming languages, frameworks, libraries, and tools. This chapter provides basic introduction of TypeScript, Angular 2, RxJS, Sass, Jasmine, and Karma. You may still need to refer to other online resources to gain further understanding. In the next chapter, we'll go through the skeleton code generated by Ionic CLI to see how Ionic 2 apps are organized.

Chapter 3

Basic App Structure

After we have prepared the local Ionic development environment and learned the basic of developing Ionic apps, it's time to dive into the details of Ionic framework. To better explain Ionic framework, we need to build a good sample app. In this book, we are going to create a mobile app for the popular Hacker News using Ionic.

If you search for Hacker News apps in major app stores, you can find a lot of existing apps. But this book still uses Hacker News app as the sample for the following reasons:

- Hacker News is very popular in the community, so it's easy for developers to understand what this app is for. We don't need to explain the background for the app.
- This app represents many content-centric mobile apps that are suitable for building with Ionic. What we discussed in this book can be applied in real product development.
- Hacker News has a Firebase-based API to retrieve its data, so it's a good example of both Ionic and Firebase.

To describe the app's requirements, we list the main user stories as below.

- View top stories – Users can view a list of top stories on Hacker News and view the page of each story.
- View comments – For each story, users can view its comments. For each comment, users can also view replies to that comment.

- Add stories to favorites – Users can add a story to favorites and see a list of all favorite stories.
- Share stories – Users can share stories to the social network.

In the following chapters, we are going to implement these user stories.

Understanding the Basic App Structure

Once we are clear about the user stories that need to be implemented in the app, we can start building the app.

An Ionic 2 app typically starts from a starter template. For this app, we choose `sidemenu` as the starter template. We use the following command to create the basic code structure of this app. The app name is `hacker_news_app_v2`.

```
$ ionic start hacker_news_app_v2 sidemenu --v2
```

Then we add the support for both iOS and Android platforms.

```
$ ionic platform add ios --save  
$ ionic platform add android --save
```

After the basic code is generated, we'll go through the code base first to get a rough idea about how source code is organized in Ionic 2 apps. This is a very important step for us to know where to add new code.

Since Ionic is built on top of Apache Cordova, an Ionic app is also a Cordova app, so some of the app's code is related to Cordova.

Config Files

At the root directory of the project, we can find several config files. These config files are used by different libraries or tools.

package.json

`package.json` is a JSON file that describes this Node.js project. This file contains various metadata of this project, including name, description, version, license, and other information. This file is also used by npm or yarn to manage a project's dependencies.

`package.json` also contains metadata used by Cordova. For example, `cordovaPlugins` specifies installed plugins for this app, while `cordovaPlatforms` specifies supported platforms.

The content of `package.json` file can be managed by tools like npm, yarn, or Ionic CLI, or edited manually using text editors.

config.xml

`config.xml` is the configuration file for Apache Cordova. More information about this file can be found at the Apache Cordova website (http://cordova.apache.org/docs/en/dev/config_ref/index.html#The%20config.xml%20File). The content of this file is usually managed by Cordova CLI.

tsconfig.json

`tsconfig.json` in Listing 3-1 is the JSON file to configure how TypeScript compiler compiles the TypeScript code into JavaScript. For example, `include` and `exclude` specify the files to include or exclude for the compiler, respectively. `compilerOptions` specifies the compiler options. In these compiler options, `emitDecoratorMetadata` and `experimentalDecorators` must be enabled to use Angular 2 decorators.

Listing 3-1. tsconfig.json file

```
{
  "compilerOptions": {
    "allowSyntheticDefaultImports": true,
    "declaration": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [
      "dom",
      "es2015"
    ],
    "module": "es2015",
    "moduleResolution": "node",
    "sourceMap": true,
    "target": "es5"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ],
  "compileOnSave": false,
  "atom": {
    "rewriteTsconfig": false
  }
}
```

ionic.config.json

ionic.config.json in Listing 3-2 is the config file for Ionic itself. From this file, we can see the app is an Ionic 2 project with TypeScript. The app_id property is used with Ionic Cloud. The app is currently not using features from Ionic Cloud, so app_id is an empty string.

Listing 3-2. ionic.config.json file

```
{
  "name": "hacker_news_app_v2",
  "app_id": "",
  "v2": true,
  "typescript": true
}
```

tslint.json

tslint.json is the config file for TSLint. Ionic 2 provides a default configuration. More rules (<https://palantir.github.io/tslint/rules/>) can be added to match the development team's style guide.

.editorconfig

.editorconfig is the config file (<http://editorconfig.org/>) to define coding styles for different IDEs and editors.

Cordova Files

Besides the config.xml in the root directory, hooks, platforms, plugins and www directories are all managed by Cordova.

hooks

hooks directory contains Cordova Hooks that are special scripts to customize Cordova commands.

platforms

For each supported platform, there is a subdirectory in platforms to contain built files for this platform. This app has ios and android subdirectories for iOS and Android platforms, respectively.

plugins

This directory contains various Cordova plugins used in this app.

www

The `www` directory contains static files of this app. This is also the directory of the whole app's static files, including built JavaScript and CSS files.

App Files

All the app's main source code is in the `src` directory. The `resources` directory contains the image files for the app icons and the splash screen.

index.html

As we know, an Ionic app is just a web app running inside the browser-like `WebView` component. The `index.html` is the entry point of the whole Ionic app. Like normal HTML pages, `index.html` contains HTML markups and references to JavaScript and CSS files. Ionic uses TypeScript and Sass, but `index.html` only refers to the built JavaScript and CSS files.

declarations.d.ts

The file `declarations.d.ts` contains type definitions for third-party libraries used in the app. If the third-party libraries you are trying to use don't have type definitions available in the TypeScript community registry, you can add them in this file.

manifest.json and service-worker.js

These two files are related to creating Progressive Web Apps using Ionic and are out of this book's scope.

assets

The `assets` directory contains various assets used in the app, including favicon and fonts.

theme

The theme directory contains Sass files to customize the look and feel of the app.

app

The app directory contains the app modules and the root component.

components

The components directory contains other components declared in the app. This directory doesn't exist in the sidemenu template code but should exist in any nontrivial apps.

pages

The pages directory contains code for different pages. Each page has its own subdirectory.

Skeleton Code

The sidemenu template already includes some basic code. It has two demo pages and the side menu has links to both pages. Clicking the menu item opens the corresponding page.

app.module.ts

The `app.module.ts` file in Listing 3-3 declares the root module of the app. This file contains only a single empty class `AppModule`. Most of the code is using `NgModule` decorator to annotate the class `AppModule`. The imported module `IonicModule.forRoot(MyApp)` in the array of the property imports is the difference between normal Angular 2 modules and Ionic modules. The `IonicModule.forRoot()` method wraps a root component to make sure that the components and directives from Ionic framework are provided when the root component is loaded. These components and directives can be used anywhere in the app. The `IonicApp` in the array of the property bootstrap is to bootstrap the Ionic app. The properties declarations and entryComponents both contain the three components `App`, `Page1`, and `Page2` included in the app. There are no services in the app yet, so the value of the property providers is an empty array.

Listing 3-3. app.module.ts

```

import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Page1 } from '../pages/page1/page1';
import { Page2 } from '../pages/page2/page2';
@NgModule({
  declarations: [
    MyApp,
    Page1,
    Page2
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    Page1,
    Page2
  ],
  providers: []
})
export class AppModule {}

```

app.component.ts

The `app.component.ts` file in Listing 3-4 declares the main component of the app. In the Component decorator, only the property `templateUrl` is set. `Nav` is used for page navigation and will be discussed in later chapters. The constructor function declares a public parameter `platform` of type `Platform`. Angular 2 injector creates the instance of `Platform` and provides it when this component is instantiated. The `ready()` method of `Platform` returns a promise that resolved when the Cordova platform is ready to use. Here it set the styles of the status bar and hide the splash screen. The property `pages` is an array of pages included in the skeleton code. The `openPage()` method is invoked by the menu items to open different pages.

Listing 3-4. app.component.ts

```

import { Component, ViewChild } from '@angular/core';
import { Nav, Platform } from 'ionic-angular';
import { StatusBar, SplashScreen } from 'ionic-native';
import { Page1 } from '../pages/page1/page1';
import { Page2 } from '../pages/page2/page2';
@Component({
  templateUrl: 'app.html'
})

```

```
export class MyApp {
  @ViewChild(Nav) nav: Nav;
  rootPage: any = Page1;
  pages: Array<{title: string, component: any}>;
  constructor(public platform: Platform) {
    this.initializeApp();
    this.pages = [
      { title: 'Page One', component: Page1 },
      { title: 'Page Two', component: Page2 }
    ];
  }
  initializeApp() {
    this.platform.ready().then(() => {
      StatusBar.styleDefault();
      SplashScreen.hide();
    });
  }
  openPage(page) {
    this.nav.setRoot(page.component);
  }
}
```

app.html

The template file `app.html` in Listing 3-5 includes two Ionic components: the `ion-menu` to show the side menu and the `ion-nav` as the container of different pages. Directives like `ion-toolbar` and `ion-list` will be covered in later chapters.

Listing 3-5. app.html

```
<ion-menu [content]="content">
  <ion-header>
    <ion-toolbar>
      <ion-title>Menu</ion-title>
    </ion-toolbar>
  </ion-header>
  <ion-content>
    <ion-list>
      <button menuClose ion-item *ngFor="let p of pages"
(click)="openPage(p)">
        {{p.title}}
      </button>
    </ion-list>
  </ion-content>
</ion-menu>

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

main.ts

The `main.ts` file in Listing 3-6 contains the logic to bootstrap the Ionic app.

Listing 3-6. main.ts

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

app.scss

The `app.scss` file contains additional CSS styles. If you don't need to customize the theme of Ionic 2, just leave this file empty.

Page 1 and Page 2 Files

The components `Page1` and `Page2` have their own subdirectories under the directory `pages`. Each component has three files: TypeScript file for the component class, HTML file as the view template, and Scss file for styles. This is the common code convention when developing components in Ionic 2.

Summary

In this chapter, we discuss the user stories of the example Hacker News app and use Ionic CLI to generate the skeleton code of the app. We go through the skeleton code to understand how the code of an Ionic 2 app is organized. This skeleton code is the starting point of the example app. In the next chapter, we'll see how to implement the first user story of listing top stories on Hacker News.

Chapter 4

List Stories

Based on the existing `sidemenu` template code, we start implementing the first user story that lists Hacker News top stories. Because this is the first chapter about implementing user stories, some common concepts are introduced in this chapter. We are going to cover the following topics in this long chapter.

- Use the list component to show top stories and test with Jasmine and Karma.
- Services to load top stories.
- Firebase basics and Hacker News API.
- Infinite scrolling and pull-to-refresh.
- Loading indicators and error handling.

This chapter also shows how to gradually improve the implementation of a user story to make it more user friendly and reliable, which is a recommended practice in actual product development. We start from a simple list with dummy data and improve it to a full-fledged component with real data.

Define the Model

As we mentioned before, Hacker News has a public API to list top stories, which is a public Firebase database. It's a bad idea to jump directly into the implementation details. We should start from the user experience we want to provide.

After the app starts, the user is presented with a list of top stories on Hacker News. The user can see basic information of each story in the list, including title, URL, author, published time, and score. The information for each story should be declared in the model. Each story has these properties: `id`, `title`,

url, by, time and score. We declare the model as a TypeScript interface in Listing 4-1. Here we use a more general model name `Item` instead of `Story` because comments are also items and can share the same model.

Listing 4-1. Item model

```
export interface Item {  
  id: number;  
  title: string;  
  url: string;  
  by: string;  
  time: number;  
  score: number;  
}
```

The list of top stories can be represented as an array of items, that is, `Item[]`. However, we define a type `Items` in Listing 4-2 to represent a list of items. Even though `Items` is currently just an alias of `Item[]`, declaring a dedicated type `Items` enables potential future improvements without changing existing code. For example, the list of items may only represent a paginated subset of all items, so properties related to pagination like `limit`, `offset` or `total` may need to be added to the model `Items` in the future. All the existing code can still use the same type `Items`.

Listing 4-2. Items model

```
import { Item } from './Item';  
  
export type Items = Item[];
```

List Component

After we have a list of items, we need to find a UI component to display them. We can use the Ionic list component. The list component is a generic container of an array of items. Items in the list can be text only or text with icons, avatars, or thumbnails. Items can be a single line or multiple lines. We can use the list component to display top stories. Let's start with the introduction of the list component.

Simple List

Listing 4-3 shows a simple list with three text-only items. The component `ion-list` is the container of items, while the component `ion-item` represents a single item in the list.

Listing 4-3. Simple list

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

Header and Separators

A list can contain a header to display a title at the top of the list. The header is created by adding the component `ion-list-header` as the child of the `ion-list`. Items in the list can be separated using the component `ion-item-divider`.

In Listing 4-4, a title is added to the list and a divider separates two items.

Listing 4-4. List with header and dividers

```
<ion-list>
  <ion-list-header>
    Items
  </ion-list-header>
  <ion-item>
    Item 1
  </ion-item>
  <ion-item-divider>
  </ion-item-divider>
  <ion-item>
    Item 2
  </ion-item>
</ion-list>
```

Grouping of Items

If a list contains a lot of items, it's better to group those items for users to view and find them easily. A typical example is the contact list, which is usually grouped by the first character of each contact's first name. Grouping of items can be done using the component `ion-item-group`.

Listing 4-5 is an example of the simple contacts list. We use the component `ion-item-divider` to show the name of each group. We don't need to use the component `ion-list` when grouping items.

Listing 4-5. Grouping of items

```
<ion-item-group>
  <ion-item-divider>A</ion-item-divider>
  <ion-item>Alex</ion-item>
  <ion-item>Amber</ion-item>
</ion-item-group>
<ion-item-group>
  <ion-item-divider>B</ion-item-divider>
  <ion-item>Bob</ion-item>
  <ion-item>Brenda</ion-item>
</ion-item-group>
```

Icons

Ionic 2 provides over 700 ready-to-use icons. These icons can be displayed using the component `ion-icon` with the name of the icon. For each icon, there may be three variations for different platforms, that is, iOS, iOS-Outline, and Material Design. For example, the icon `book` has three variations with names `ios-book`, `ios-book-outline`, and `md-book`. If the name `book` is used, the displayed icon is auto-selected based on the current platform. We can also use the property `ios` and `md` to explicitly set the icon for different platforms. An icon can also be set to be active or inactive by setting the property `isActive`.

In Listing 4-6, the first `ion-item` has the icon `book` to the left of the text. The second `ion-item` has the icon `build` at both sides of the text, but the icon on the left side is inactive. The last `ion-item` shows the icon `ios-happy` on iOS, but shows the icon `md-sad` on Android.

Listing 4-6. List with icons

```
<ion-list>
  <ion-item>
    <ion-icon name="book" item-left></ion-icon>
    Book
  </ion-item>
  <ion-item>
    <ion-icon name="build" isActive="false" item-left></ion-icon>
    Build
    <ion-icon name="build" item-right></ion-icon>
  </ion-item>
  <ion-item>
    <ion-icon ios="ios-happy" md="md-sad" item-right></ion-icon>
    Happy or Sad
  </ion-item>
</ion-list>
```

Avatars

Avatars create circular images. They are typically used to display users' profile images. In Ionic 2, avatars are created by using the component `ion-avatar` to wrap `img` elements.

Listing 4-7 shows a list of three items that have avatars.

Listing 4-7. List with avatars

```
<ion-list>
  <ion-item>
    <ion-avatar item-left>
      
    </ion-avatar>
    Alex
  </ion-item>
  <ion-item>
    <ion-avatar item-left>
      
    </ion-avatar>
    Bob
  </ion-item>
  <ion-item>
    <ion-avatar item-left>
      
    </ion-avatar>
    David
  </ion-item>
</ion-list>
```

Thumbnails

Thumbnails are rectangular images larger than avatars. In Ionic 2, thumbnails are created by using the component `ion-thumbnail` to wrap `img` elements.

Listing 4-8 shows a list of three items that have thumbnails.

Listing 4-8. List with thumbnails

```
<ion-list>
  <ion-item>
    <ion-thumbnail item-left>
      
    </ion-thumbnail>
    Apple
  </ion-item>
```

```
<ion-item>
  <ion-thumbnail item-left>
    
  </ion-thumbnail>
  Banana
</ion-item>
<ion-item>
  <ion-thumbnail item-left>
    
  </ion-thumbnail>
  Orange
</ion-item>
</ion-list>
```

In Listings 4-6, 4-7, and 4-8, the property `item-left` or `item-right` is used in the components `ion-icon`, `ion-avatar` or `ion-thumbnail` to place the icon, avatar, or thumbnail to the left or right side of the item. There is a difference between `item-left` and `item-right` regarding the element's position. The element with `item-left` is placed outside of the main item wrapper, but the element with `item-right` is placed inside of the main item wrapper. This is made clear by checking the position of the bottom line that separates each item.

Display a List of Items

After defining the model `Item` and learning the component `ion-list`, we are now ready to display a list of items in the app. We need to create a component for the list of items and another component for each item in the list. The easiest way to create new components is by using Ionic CLI. We use the commands below to create two new components. The `g` command is short for `generate`. `g component` means creating components.

```
$ ionic g component Items
$ ionic g component Item
```

Ionic CLI creates the necessary files for a component, including TypeScript file, HTML template file, and Sass file. For the generated component, these files are created under the related subdirectory of the `src/components` directory. For example, files of the `Items` component, `items.ts`, `items.html` and `items.scss`, are created in the directory `src/components/items`. Ionic CLI also adds the suffix `Component` to the class name of the generated component. The name of the component `Items` is `ItemsComponent`.

Item Component

Let's start from the TypeScript file `item.ts` of the component `ItemComponent`. In Listing 4-9, class `ItemComponent` is decorated with `Component`. The value of the property selector is `item`, which means that this component can be created using the element `item` in the template, for example, `<item></item>`. The `Input` decorator factory of the class instance property `item` means the value of this property is bound to the value of the property `item` in its parent component. The `Input` decorator factory can take an optional argument that specifies a different name of the bound property in the parent component, for example, `@Input('myItem')` binds to the property `myItem` in the parent component. Type of the property `item` is the model interface `Item` we defined in Listing 4-1.

Listing 4-9. Item component

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

@Component({
  selector: 'item',
  templateUrl: 'item.html',
})
export class ItemComponent {
  @Input() item: Item;
}
```

Listing 4-10 shows the template file `item.html` of the component `Item`. In the template, we can use `item.title`, `item.score`, `item.by`, `item.time` and `item.url` to access properties of the current bound item. The layout of the item is simple. The title is displayed in a `h2` element. The score, author, and published time are displayed in the same row after the title. The URL is displayed at the bottom.

Listing 4-10. Template of the item component

```
<div>
  <h2 class="title">{{ item.title }}</h2>
  <div>
    <span>
      <ion-icon name="bulb"></ion-icon>
      {{ item.score }}
    </span>
    <span>
      <ion-icon name="person"></ion-icon>
      {{ item.by }}
    </span>
    <span>
      <ion-icon name="time"></ion-icon>
```

```
        {{ item.time | timeAgo }}
      </span>
    </div>
  <div>
    <span>
      <ion-icon name="link"></ion-icon>
      {{ item.url }}
    </span>
  </div>
</div>
```

The `timeAgo` used in `{{ item.time | timeAgo }}` is an Angular 2 pipe to transform the timestamp `item.time` into a human-readable text, like 1 minute ago or 5 hours ago. The implementation of `TimeAgoPipe` uses the [taijinlee/humanize](https://github.com/taijinlee/humanize) (<https://github.com/taijinlee/humanize>) library to generate a human-readable text.

In Listing 4-11, the decorator factory `Pipe` annotates the class `TimeAgoPipe` to be a pipe. In the method `transform()` of `PipeTransform`, we use the method `relativeTime()` from the `humanize` library to convert the input time to a string.

Listing 4-11. timeAgo pipe

```
import { Pipe, PipeTransform } from '@angular/core';
import * as humanize from 'humanize';

@Pipe({ name: 'timeAgo' })
export class TimeAgoPipe implements PipeTransform {
  transform(time: number) {
    return humanize.relativeTime(time);
  }
}
```

The `item.scss` file in Listing 4-12 contains the basic style of the component `Item`. All the CSS style rules are scoped in the element selector `item` to make sure that styles defined for this component won't conflict with styles from other components.

Listing 4-12. Styles of the item component

```
item {
  .title {
    font-size: 1.8rem;
  }

  .link {
    font-size: 1.2rem;
  }
}
```

```

    div > span:not(:last-child) {
      padding-right: 1rem;
    }
  }
}

```

Items Component

The `ItemsComponent` is used to render a list of items. In Listing 4-13 of its main TypeScript file `items.ts`, the selector for this component is `items`. It also has the property `items` that are bound to the value of the property `items` from its parent component.

Listing 4-13. Items component

```

import { Component, Input } from '@angular/core';
import { Items } from '../model/Items';

@Component({
  selector: 'items',
  templateUrl: 'items.html',
})
export class ItemsComponent {
  @Input() items: Items;
}

```

In the template file `items.html` shown in Listing 4-14, we use the component `ion-list` as the container. We use the directive `ngFor` to loop through the array of items. For each item in the `items` list, a component `ion-item` is created to wrap an item component.

Listing 4-14. Template of items component

```

<ion-list>
  <ion-item *ngFor="let item of items">
    <item [item]="item"></item>
  </ion-item>
</ion-list>

```

Empty List

If a list contains no items, it should display a message to inform the user. We can use the `ngIf` directive to display different views based on the length of the list; see Listing 4-15.

Listing 4-15. Show empty list

```
<ion-list *ngIf="items.length > 0">
  <ion-item *ngFor="let item of items">
    <item [item]="item"></item>
  </ion-item>
</ion-list>
<p *ngIf="items.length === 0">
  No items.
</p>
```

Test List Component

Even though the component `ItemsComponent` is very simple so far, it's still a good practice to add tests for this component. The project created using Ionic starter template doesn't have testing support, so we need to set up the basic tools for testing. We usually need two kinds of tests:

- Unit tests – Test a single unit, for example, a component, a service, or a pipe.
- End-to-end tests – Test a user story, usually for testing integration between different units.

So far, we only need to add unit tests for the component.

Tools

We'll use Jasmine and Karma for testing, which are both used in standard Angular 2 projects. Jasmine is used to write test specs, while Karma is used to run tests on different browsers.

To configure the testing support for Ionic projects, we use yarn to add Jasmine and Karma related modules as dev dependencies.

```
$ yarn add --dev jasmine-core karma karma-chrome-launcher karma-cli
karma-htmlfile-reporter karma-jasmine karma-jasmine-html-reporter
karma-mocha-reporter
```

If an Angular 2 app is created using the official quickstart (<https://angular.io/docs/ts/latest/quickstart.html>) template, then the testing infrastructure has been configured properly. To add testing support to our existing Ionic app, we can copy the testing configuration from the quickstart template code and merge into the Ionic app. However, this approach requires a lot of customization. A better approach is using the Angular 2 CLI (<https://cli.angular.io/>).

The goal of Angular 2 CLI is to easily create Angular 2 apps that work out of the box and follow best practices from the Angular 2 development team. After the `angular-cli` package is installed using `yarn`, we can use commands like `ng new` to create new Angular 2 apps, or `ng generate` to create components, directives, pipes, or services. For our Ionic app, we can use `ng test` to run unit tests.

```
$ yarn add --dev angular-cli
```

Testing Configuration

Our Ionic app is not created using Angular 2 CLI, so we need to add the `angular-cli.json` in the root directory to make Angular 2 CLI recognize it as an Angular 2 app. In Listing 4-16, the property `apps` only contains a single JavaScript object that configures the current Ionic app as an Angular 2 app.

Listing 4-16. angular-cli.json

```
{
  "project": {
    "version": "1.0.0",
    "name": "hacker news"
  },
  "apps": [
    {
      "root": "src",
      "outDir": "dist",
      "assets": "assets",
      "index": "index.html",
      "main": "main.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.test.json",
      "prefix": "app",
      "mobile": false,
      "scripts": [],
      "environments": {
        "source": "environments/environment.ts",
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
      }
    }
  ],
  "addons": [],
  "packages": [],
```

```
"e2e": {
  "protractor": {
    "config": "../protractor.conf.js"
  }
},
"test": {
  "karma": {
    "config": "../karma.conf.js"
  }
},
"defaults": {
  "prefixInterfaces": false
}
}
```

The property `test` in Listing 4-16 configures Karma to use the file `karma.conf.js` in Listing 4-17. All the other configurations are not important because we only use Angular CLI to run tests.

Listing 4-17. karma.conf.js

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', 'angular-cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-mocha-reporter'),
      require('karma-jasmine-html-reporter'),
      require('karma-htmlfile-reporter'),
      require('angular-cli/plugins/karma')
    ],
    files: [
      { pattern: '../src/karma-test-shim.ts', watched: false },
    ],
    preprocessors: {
      '../src/karma-test-shim.ts': ['angular-cli'],
    },
    angularCli: {
      config: '../angular-cli.json',
      environment: 'dev',
    },
    htmlReporter: {
      outputFile: '_test-output/unit-tests/index.html',
      pageTitle: 'Unit Tests',
      subPageTitle: __dirname,
    },
  },
}
```

```

reporters: [
  'mocha', 'html', 'kjhtml',
],
port: 9876,
colors: true,
logLevel: config.LOG_INFO,
autoWatch: true,
browsers: ['Chrome'],
singleRun: false,
mime: {
  'text/x-typescript': ['ts', 'tsx'],
},
});
};

```

The file `karma.conf.js` has different configurations.

- `basePath` – The base path to resolve other relative paths.
- `frameworks` – Frameworks to use. Here we use Jasmine and Angular CLI.
- `plugins` – Plugins for Karma. Here we use plugins for Jasmine, Chrome launcher, Angular CLI, and different reporters.
- `files` – Files to load when running the tests. Only one file `./src/karma-test-shim.ts` is specified.
- `preprocessors` – Preprocessors to process files before loaded by Karma. Here we use Angular CLI to process the entry point file to compile it to JavaScript for browsers to load.
- `angularCli` – Configurations for the Angular CLI plugin.
- `htmlReporter` – Configurations for the HTML reporter.
- `reporters` – Names of all reporters.
- `port`: Server port.
- `colors` – Show colors in the console.
- `logLevel` – Set the log level to `INFO`.
- `autoWatch` – Watch for file changes and rerun tests.
- `browsers` – Browsers to launch. Here we only use Chrome.
- `singleRun` – Whether Karma should shut down after the first run. Here it's set to `false`, so Karma keeps running tests after file changes.

- **mime** – Configure extra MIME types. Here we configure the MIME types for TypeScript files. This is important for Chrome and Firefox to correctly load TypeScript files. Otherwise, Karma cannot find any tests to run due to the error: Refused to execute script from 'http://localhost:9876/base/src/karma-test-shim.ts' because its MIME type ('video/mp2t') is not executable.

The file `karma-test-shim.ts` in Listing 4-18 is the entry point to bootstrap the testing. After Angular 2 core testing modules are imported, we initialize the test bed for component testing, then start the testing. All test spec files with suffix `.spec.ts` are loaded and executed.

Listing 4-18. karma-test-shim.ts

```
import 'core-js/es6';
import 'core-js/es7/reflect';

import 'zone.js/dist/zone';
import 'zone.js/dist/long-stack-trace-zone';
import 'zone.js/dist/proxy.js';
import 'zone.js/dist/sync-test';
import 'zone.js/dist/jasmine-patch';
import 'zone.js/dist/async-test';
import 'zone.js/dist/fake-async-test';

declare var __karma__: any;
declare var require: any;

__karma__.loaded = function (): any { };

Promise.all([
  System.import('@angular/core/testing'),
  System.import('@angular/platform-browser-dynamic/testing'),
])
  .then(([testing, testingBrowser]) => {
    testing.getTestBed().initTestEnvironment(
      testingBrowser.BrowserDynamicTestingModule,
      testingBrowser.platformBrowserDynamicTesting()
    );
  })
  .then(() => require.context('.', true, /\.spec\.ts/))
  .then(context => context.keys().map(context))
  .then(__karma__.start, __karma__.error);
```

Testing Items Component

Now we can add a test suite for the items component. A test suite file uses the same name as the component it describes, but with a different suffix `.spec.ts`. For example, `items.spec.ts` in Listing 4-19 is the test suite file for the items component with name `items.ts`.

Listing 4-19. items.spec.ts

```
import { ComponentFixture, async } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { TestUtils } from '../test';
import { Item } from '../model/Item';
import { Items } from '../model/Items';
import { ItemsComponent } from './items';
import { ItemComponent } from './item/item';
import { TimeAgoPipe } from '../pipes/TimeAgoPipe';

let fixture: ComponentFixture<ItemsComponent> = null;
let component: any = null;

describe('items component', () => {

  beforeEach(async(() => TestUtils.beforeEachCompiler([ItemsComponent,
    ItemComponent, TimeAgoPipe])).then(compiled => {
    fixture = compiled.fixture;
    component = compiled.instance;
  }));

  it('should display a list of items', () => {
    const results: Item[] = [{
      id: 1,
      title: 'Test item 1',
      url: 'http://www.example.com/test1',
      by: 'user1',
      time: 1478576387,
      score: 242,
    }, {
      id: 2,
      title: 'Test item 2',
      url: 'http://www.example.com/test2',
      by: 'user2',
      time: 1478576387,
      score: 100,
    }];
    const items: Items = {
      offset: 0,
      limit: results.length,
```

```
        total: results.length,
        results,
    });
    component.items = items;
    fixture.detectChanges();
    let debugElements = fixture.debugElement.queryAll(By.css('h2'));
    expect(debugElements.length).toBe(2);
    expect(debugElements[0].nativeElement.textContent).toContain('Test item 1');
    expect(debugElements[1].nativeElement.textContent).toContain('Test item 2');
  });

  it('should display no items', () => {
    component.items = {
      offset: 0,
      limit: 0,
      total: 0,
      results: [],
    };
    fixture.detectChanges();
    let debugElement = fixture.debugElement.query(By.css('p'));
    expect(debugElement).not.toBeNull();
    expect(debugElement.nativeElement.textContent).toContain('No items');
  });
});
```

In Listing 4-19, we use the method `describe()` to create a new test suite. The method `beforeEach()` is used to add code to execute before execution of each spec. `TestUtils` used in `beforeEach()` is an important utility class; see Listing 4-20. The method `beforeEachCompiler()` takes a list of components and providers as the input arguments and uses the method `configureIonicTestingModule()` to configure the testing module. The testing module imports Ionic modules and adds Ionic providers. This is required to set up the runtime environment for Ionic. After the testing module is configured, the components are compiled. After the compilation, the method `TestBed.createComponent()` creates an instance of the first component and returns the component test fixture. In the test suite, the returned test fixture and component instance are used for verifications.

Listing 4-20. TestUtils

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { TestBed } from '@angular/core/testing';
import { App, MenuController, NavController, GestureController, Platform,
Config, Keyboard, Form, IonicModule } from 'ionic-angular';
import { ConfigMock, DomControllerMock, PlatformMock } from './mocks';
```

```

export class TestUtils {

    public static beforeEachCompiler(components: Array<any>, providers:
Array<any> = []): Promise<{fixture: any, instance: any}> {
        return TestUtils.configureIonicTestingModule(components, providers)
            .compileComponents().then(() => {
                let fixture: any = TestBed.createComponent(components[0]);
                return {
                    fixture,
                    instance: fixture.componentInstance,
                };
            });
    }

    public static configureIonicTestingModule(components: Array<any>,
providers: Array<any> = []): typeof TestBed {
        return TestBed.configureTestingModule({
            declarations: [
                ...components,
            ],
            providers: [
                App,
                { provide: Config, useClass: ConfigMock },
                { provide: DomController, useClass: DomControllerMock },
                Form,
                Keyboard,
                MenuController,
                NavController,
                GestureController,
                { provide: Platform, useClass: PlatformMock },
                ...providers,
            ],
            imports: [
                FormsModule,
                IonicModule,
                ReactiveFormsModule,
            ],
        });
    }
}

```

In the class `TestUtils`, we use some mock classes; see Listing 4-21. This is because some Ionic built-in modules require runtime information that is not available in the testing environment. These mock classes are used to work around JavaScript errors in those modules when running the tests.

Listing 4-21. Mocks

```
import { Platform } from 'ionic-angular';

export class ConfigMock {
  public get(): any {
    return '';
  }

  public getBoolean(): boolean {
    return true;
  }

  public getNumber(): number {
    return 1;
  }
}

export class DomControllerMock {
  public read(): any {}
  public write(): any {}
}

export class PlatformMock extends Platform {
  public ready(): any {
    return new Promise((resolve: Function) => {
      resolve();
    });
  }

  public registerListener(): any {}

  public getActiveElement(): any {
    return null;
  }
}
```

In Listing 4-19, we created two test specs. The first spec is to test that the list component can display a list of items. The array `items` contains two items, so there should be two rendered `h2` elements for items' titles and the text should match the corresponding item's value of the property `title`. The `fixture.debugElement.queryAll()` method queries all the elements matching the given predicate. The `By.css()` method creates a predicate using a CSS selector. We can use the property `nativeElement` to get the native DOM element from the query result. The second spec is to test the list component displays correct message when no items in the list. The property `items` is set to an empty array, so there should be a `p` element with text `No items`.

Run Tests

To run tests using Karma, we add a new script test to `package.json` with content `ng test`. Then we can use `yarn run test` to start the testing. Karma launches the Chrome browser and runs the tests inside of the browser. The testing results will be reported in the console. Figure 4-1 shows the UI of Karma running on Chrome.

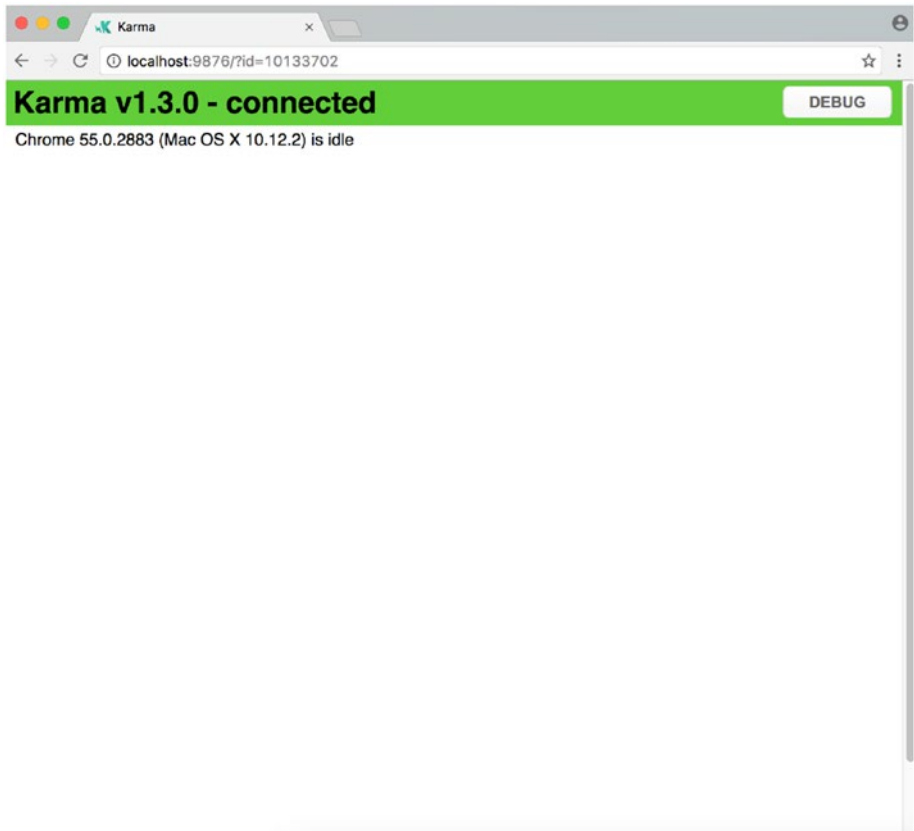


Figure 4-1. Karma UI

Clicking the **DEBUG** button in Figure 4-1 shows the debug page; see Figure 4-2. In this page, we can see the results of all specs. We can also use Chrome Developer Tools to debug errors in the test scripts.

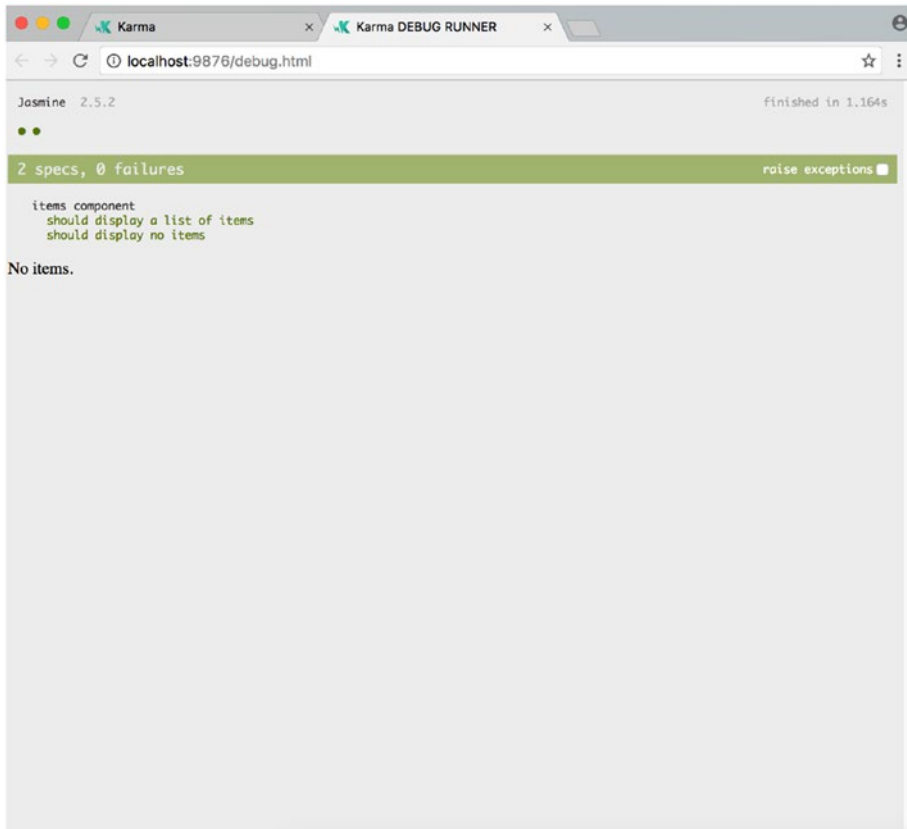


Figure 4-2. Karma debug UI

Items Loading Service

When testing the items list component, we use the hard-coded values for the array `items`. This is good for testing since we can easily verify the test results. Now we are going to use real data to render the items list. In Angular 2 apps, code logic that deals with external resources should be encapsulated in services.

`ItemService` in Listing 4-22 has a single method `load(offset, limit)` to load a list of items. Because there can be many items, the `load()` method only loads a subset of items. Parameters `offset` and `limit` are used for pagination: `offset` specifies the position of the first loaded item in the whole items list, `limit` specifies the number of loaded items. For example, `load(0, 10)` means loading the first 10 items.

Listing 4-22. ItemService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Items } from '../model/Items';

@Injectable()
export default class ItemService {
  load(offset: number, limit: number): Observable<Items> {
    return Observable.of({
      offset: 0,
      limit: 0,
      total: 0,
      results: [],
    });
  }
}
```

The return type of the `load()` method is `Observable<Items>`. `Items` is the model type we defined in Listing 4-2. It used to be just an alias of `Item[]`, now we update it to include more information related to pagination, see Listing 4-23. The `Items` type now has both `offset` and `limit` properties that match the parameters in the `load()` method. It also has the property `total` that represents the total number of items. The property `total` is optional, because in some cases the total number may not be available. The property `results` represents the actual array of items.

Listing 4-23. Updated Item model

```
import { Item } from './Item';

export interface Items {
  offset: number;
  limit: number;
  total?: number;
  results: Item[];
}
```

After we updated the model `Items`, we also need to update code in the items list component and related test specs to use the new model.

`ItemService` uses decorator factory `Injectable`, so it'll be registered to Angular 2's injector and available for dependency injection to other components.

The return `Observable<Items>` of the `load()` method currently only emits an `Items` object with empty results. We'll improve the implementation to use the actual Hacker News API later.

Top Stories Page

Now we can create a new page in the Ionic app to show top stories using `ItemsComponent` and `ItemService`. This page is also the new index page.

We can create a new page using Ionic CLI. After executing the command below, stub code of the new page is created in the directory `pages/top-stories`.

```
$ ionic g page topStories
```

In Listing 4-24, the page class `TopStoriesPage` implements the Angular 2 interface `OnInit`. By implementing this interface, when `TopStoriesPage` page finishes initialization, the `ngOnInit()` method will be invoked automatically. The constructor of class `TopStoriesPage` takes two arguments, `navCtrl` of type `NavController` and `itemService` of type `ItemService`. `NavController` is provided by Ionic for page navigation. We'll discuss page navigation in Chapter 6, so we can ignore `navCtrl` now. We use `ItemService` to load items and use `ItemsComponent` to display them.

`TopStoriesPage` class has a property `items` of type `Items`. The value of this property is passed to the `ItemsComponent` for rendering. In the `ngOnInit()` method, the method `load()` of `ItemService` is invoked. When the loading is finished, loaded items is set as the value of the property `items`. `TopStoriesPage` class also implements the `OnDestroy` interface. In the method `ngOnDestroy()`, we use the method `unsubscribe()` of the `Observable` subscription to make sure resources are released.

Listing 4-24. top-stories.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { NavController } from 'ionic-angular';
import { Subscription } from "rxjs";
import { Items } from '../model/Items';
import { ItemService } from '../services/ItemService';

@Component({
  selector: 'page-top-stories',
  templateUrl: 'top-stories.html'
})
export class TopStoriesPage implements OnInit, OnDestroy {
  items: Items;
  subscription: Subscription;
  constructor(public navCtrl: NavController, private itemService:
ItemService) {}
```

```

ngOnInit(): void {
    this.subscription = this.itemService.load(0, 10).subscribe(items =>
this.items = items);
}

ngOnDestroy(): void {
    if (this.subscription) {
        this.subscription.unsubscribe();
    }
}
}

```

In the template file of Listing 4-25, `<items *ngIf="items" [items]="items"></items>` creates the `ItemsComponent` and binds the value of the property `items` to the `items` in `TopStoriesPage` class. Because loading items are asynchronous, the value of `items` is `null` until the loading is finished successfully. The usage of directive `ngIf` is to make sure the `ItemsComponent` is only created after items are loaded.

Listing 4-25. top-stories.html

```

<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Top Stories</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <items *ngIf="items" [items]="items"></items>
</ion-content>

```

Test

Now we need to add a test suite for this page.

Items Loading Service Mock

To test the top stories page, we need a service that can load items and simulate different scenarios. The best solution is to create a service mock. `ItemServiceMock` in Listing 4-26 is the mock we use for `ItemService`. The implementation of the method `load()` in `ItemServiceMock` creates items based on input arguments `offset` and `limit`.

Listing 4-26. ItemServiceMock

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import * as range from 'lodash.range';
import { Items } from '../model/Items';
import { Item } from '../model/Item';
import { ItemService } from '../services/ItemService';

@Injectable()
export class ItemServiceMock extends ItemService {
  load(offset?: number, limit?: number): Observable<Items> {
    const results: Item[] = range(offset, offset + limit).map(index => ({
      id: index,
      title: `Item ${index + 1}`,
      url: `http://www.example.com/item${index}`,
      by: `demo`,
      time: new Date().getTime() / 1000,
      score: index,
    }));
    return Observable.of({
      offset,
      limit,
      total: offset + limit,
      results,
    });
  }
}
```

Test Suite

Now we can add a test suite for `TopStoriesPage` class. In the method `beforeEach()`, when using `TestUtils.beforeEachCompiler()` method to configure the component, we need to register the `ItemServiceMock` as a provider. Here we use the Angular 2 class providers. `{provide: ItemService, useClass: ItemServiceMock}` means the registered provider uses the token `ItemService` but uses `ItemServiceMock` as the actual class. So `TopStoriesPage` is injected with an instance of class `ItemServiceMock`.

The code of test spec in Listing 4-27 is different from what's in Listing 4-19 for the spec of `ItemsComponent`. This is because `TopStoriesPage` uses an asynchronous service `ItemService` that makes testing more complicated.

At first, we need to use the method `async()` from Angular 2 to wrap the testing function. The first `fixture.detectChanges()` call triggers the invocation of the method `load()` in `ItemService`. `fixture.whenStable()` returns a promise that waits for the returned observable of `load()` method to

emit values. In the resolve callback of the promise returned by `whenStable()`, the second `fixture.detectChanges()` call triggers `ItemsComponent` to update itself using the loaded items. At last, we verify that the rendered DOM structure matches the expected result.

Listing 4-27. top-stories.spec.ts

```
import { ComponentFixture, async } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';
import { TestUtils } from '../test';
import { TopStoriesPage } from './top-stories';
import { ItemsComponent } from '../components/items/items';
import { ItemComponent } from '../components/item/item';
import { TimeAgoPipe } from '../pipes/TimeAgoPipe';
import { ItemService } from '../services/ItemService';
import { ItemServiceMock } from '../testing/ItemServiceMock';

let fixture: ComponentFixture<TopStoriesPage> = null;
let component: any = null;

describe('top stories page', () => {

  beforeEach(async(() => TestUtils.beforeEachCompiler(
    [TopStoriesPage, ItemsComponent, ItemComponent, TimeAgoPipe],
    [{provide: ItemService, useClass: ItemServiceMock}])
  ).then(compiled => {
    fixture = compiled.fixture;
    component = compiled.instance;
  })));

  it('should display a list of 10 items', async(() => {
    fixture.detectChanges();
    fixture.whenStable().then(() => {
      fixture.detectChanges();
      let debugElements = fixture.debugElement.queryAll(By.css('h2'));
      expect(debugElements.length).toBe(10);
      expect(debugElements[0].nativeElement.textContent).toContain('Item 1');
      expect(debugElements[1].nativeElement.textContent).toContain('Item 2');
    });
  }));
});
```

Now we have a top stories page that works with mock data. We continue to improve the implementation of `ItemService` to use the actual Hacker News API. The Hacker News API is a public Firebase database. We start from Firebase basics.

Firestore Basics

Firestore is now a product of Google. To create new Firestore projects, you need a Google account. Firestore projects can be created and managed using the Firestore console (<https://firebase.google.com/console/>).

Database Structure

Firestore has a unique database structure that is different from other databases. Each Firestore database is stored as a tree-like JSON object. This tree structure is very flexible for all kinds of data. You can organize the app's data in a way suits most for your app. For an e-commerce app, its database can be organized as in Listing 4-28. The property products is a map of product id to its properties. The property customers is a map of customer id to its properties. The property orders is a map of order id to its properties.

Listing 4-28. Sample Firestore database

```
{
  "products": {
    "00001": {
      "name": "iPhone 6s plus",
      "price": 613.50
    },
    "00002": {
      "name": "LG Nexus 5",
      "price": 187.99
    }
  },
  "customers": {
    "00001": {
      "firstName": "Alex",
      "lastName": "Cheng",
      "email": "alex@example.com"
    }
  },
  "orders": {
    "00001": {
      "customer": "00001",
      "items": {
        "00001": true,
        "00002": true
      },
      "shippingCost": 5.00
    }
  }
}
```

Each data value in the database has a unique path to identify it. If you think of the whole JSON object as a file directory, then the path describes how to navigate from the root to a data value. For example, the path of the price of the product with id 00001 is `/products/00001/price`. The path `/orders` represents all orders. The path is used when reading or writing data, as well as configuring security rules.

Firestore JavaScript SDK

Firestore provides different SDKs for mobile and web apps to interact with its database. Ionic apps should use the web SDK (<https://firebase.google.com/docs/web/setup>).

To demonstrate the usage of Firestore JavaScript SDK, we use the e-commerce database as the example. The Firestore web SDK is a general library for all kinds of web apps. For Angular 2 apps, `AngularFirestore` is a better choice that offers tight integration with Angular 2. In this section, we only discuss the usage of the general web SDK, which is very important to understand how Firestore works. `AngularFirestore` is used in the Ionic 2 app and will be discussed later.

Setup

We use Firestore console to create a new project first. In the **Overview** page of your newly created Firestore project, clicking **Add Firestore to your web app** shows a dialog with code ready to be copied into the HTML file.

In Listing 4-29, we set different configuration properties for various Firestore features. `PROJECT_ID` is the id of the Firestore project.

- `apiKey` – Core Firestore app.
- `authDomain` – Firestore Authentication.
- `databaseURL` – Firestore Realtime Database.
- `storageBucket` – Firestore Storage.
- `messagingSenderId` – Firestore Cloud Messaging.

Listing 4-29. Setup Firestore web SDK

```
<script src="https://www.gstatic.com/firebasejs/3.6.0/firebase.js"></script>
<script>
  // Initialize Firestore
  var config = {
    apiKey: "<API_KEY>",
    authDomain: "<PROJECT_ID>.firebaseapp.com",
    databaseURL: "https://<PROJECT_ID>.firebaseio.com",
```

```
storageBucket: "<PROJECT_ID>.appspot.com",
messagingSenderId: "<SENDER_ID>",
};
firebase.initializeApp(config);
</script>
```

Read Data

To start interacting with the Firebase database, we need to get a reference to the database itself. This is done by calling the method `database()` of the global object `firebase`. The return result is an instance of class `firebase.database.Database`.

```
let database = firebase.database();
```

Once we get a reference to the Firebase database, we can use the method `ref(path)` of the object `database` to get a reference to the data specified by a given path. The return result `ref` is an instance of class `firebase.database.Reference`.

```
let ref = database.ref('products');
```

Reading data from the Firebase database is done asynchronously using event listeners. After adding a listener of a certain event to the object `ref`, when the event is triggered, the listener will be invoked with related data. In Listing 4-30, a listener of the value event is added to the object `ref` using the method `on(eventType, callback)`. When the data of path `/products` is changed, the event listener is invoked with a data snapshot taken at the time of the event. The event listener will also be triggered when the listener is attached to get the initial data.

Listing 4-30. Reading data

```
ref.on('value', function(snapshot) {
  console.log(snapshot.val());
});
```

The value of argument `snapshot` passed to the listener handler is a `firebase.database.DataSnapshot` object. It's an immutable object with different properties and methods to retrieve data from the snapshot. It represents the snapshot of data at given path when the event occurs. This `DataSnapshot` object is only used for retrieving data. Modifying it won't change the value in the remote database. Table 4-1 shows the properties of `DataSnapshot`.

Table 4-1. Properties of DataSnapshot

Name	Description
key	Get the key of this DataSnapshot, which is the last token in the path. For example, the key of the path /products/00001 is 00001.
ref	Get the Reference object that created this DataSnapshot object.

Table 4-2 shows the methods of DataSnapshot.

Table 4-2. Methods of DataSnapshot

Name	Description
val()	Get a JavaScript value of the data; it could be a string, number, Boolean, array, or object. null means the DataSnapshot is empty.
exists()	Check if this DataSnapshot contains any data. This method should be used instead of checking val() !== null.
forEach(action)	Invoke a function action for each child. The function receives the DataSnapshot object of each child as the argument.
child(childPath)	Get a DataSnapshot object at the relative path specified by childPath.
hasChild(childPath)	Check if data exists at the relative path specified by childPath.
hasChildren()	Check if this DataSnapshot has any children.
numChildren()	Get the number of children.

After an event listener of a Reference object is added, you can use the method `off(eventType, callback)` to remove the listener. To make sure all event listeners are correctly removed, for a certain event type on a certain reference, the number of invocation times of the method `on()` should match the number of invocation times of the method `off()`. Event listeners are not inherited. Removing an event listener on the parent node won't remove event listeners registered on its child nodes. Each event listener should be removed individually. Listing 4-31 shows different ways to remove event listeners.

Listing 4-31. Remove event listeners

```
ref.off('value', valueCallback); // Remove single listener

ref.off('value'); // Remove all listeners of 'value' event

ref.off(); // Remove all listeners for all events
```

Sometimes you only need to retrieve the data once, then the method `once(eventType, callback)` is a better choice than `on()` because event listeners added using `once()` will only be triggered once and remove themselves after that. You don't need to worry about removing event listeners.

There are five different types of events to listen when reading data; see Table 4-3.

Table 4-3. Event types

Name	Trigger condition	Arguments of listener function
value	Once with initial data and every time data changes.	DataSnapshot object.
child_added	Once for each initial child and every time a new child is added.	DataSnapshot object of the child added and key of its previous sibling child.
child_removed	Every time a child is removed.	DataSnapshot of the removed child.
child_changed	Every time a child's data changes.	DataSnapshot object of the child changed and key of its previous sibling child.
child_moved	When a child moves its position due to priority change.	DataSnapshot object of the child changed and key of its previous sibling child.

Write Data

The reference object `ref` can also be used to write data. The method `set(value, onComplete)` replaces the data at the path of the current reference object. The method `set()` will override all data under that path, so be careful when using it. The method `update(values, onComplete)` selectively updates children data specified by relative paths in the value. Only children listed in the values will be replaced. The method `update()` supports updating multiple values. `onComplete` is the callback function to invoke after data changes are synchronized to the server. If you prefer to use Promise instead of callback functions, the return value of `set()` or `update()` is a `firebase.Promise` object, which is similar to the standard Promise object.

In Listing 4-32, the method `set()` replaces the whole data under path `products/00001`, but the method `update()` only updates the price field.

Listing 4-32. Writing data

```
let ref = database.ref('products');

ref.child('00001').set({
  "name": "New iPhone 6s plus",
  "price": 699.99
});

ref.child('00001').update({
  "price": 639.99
});
```

The Reference object also has the method `remove()` to remove whole data at the given path, which is the same as using `set(null)`.

If the data to store is a list of objects without meaningful business ids, you should use the method `push(value, onComplete)` to add a new object to the list. The method `push()` creates a unique value as the key of the object to add. It's not recommended to use sequence numbers as keys for objects in the list, as it causes synchronization problems in a distributed data system. The return value of the method `push()` is a new Reference object points to the newly created path. Listing 4-33 shows two different ways to use `push()`.

Listing 4-33. Pushing data to list

```
let ref = database.ref('customers');
ref.push({
  "firstName": "Bob",
  "lastName": "Lee",
  "email": "bob@example.com"
});

ref.push().set({
  "firstName": "Bob",
  "lastName": "Lee",
  "email": "bob@example.com"
});
```

Query Data

The major disadvantage of reading data using event listeners on the Reference object is that the whole data at the given path will be read. This could affect performance for large objects or arrays. Sorting also cannot be done either. To filter and sort data under a given path, we can use other methods in the Reference object that return `firebase.database.Query` objects.

sort

Table 4-4 shows the methods for sorting. Listing 4-34 shows an example of sorting the products by price.

Table 4-4. Methods for sorting

Method	Description
<code>orderByChild(path)</code>	Order by the value of specified child path.
<code>orderByKey()</code>	Order by the value of child keys.
<code>orderByValue()</code>	Order by the value of child values; only numbers, strings, and Booleans are supported.

Listing 4-34. Sort

```
let ref = database.ref('products');
ref.orderByChild('price');
```

filter

Table 4-5 shows the methods for filtering. In Listing 4-35, we sort the products by price first, then filter it to only return the first child.

Table 4-5. Methods for filtering

Method	Description
<code>limitToFirst(limit)</code>	Limit the number of returned items from the beginning of the ordered list of results.
<code>limitToLast(limit)</code>	Limit the number of returned items from the end of the ordered list of results.
<code>startAt(value, key)</code>	Set the starting point of returned items in the ordered list of results. Only items with a value greater than or equal to the specified value or key will be included.
<code>endAt(value, key)</code>	Set the ending point of returned items in the ordered list of results. Only items with a value less than or equal to the specified value or key will be included.
<code>equalTo(value, key)</code>	Set the value to match for returned items in the ordered list of results. Only items with a value equal to the specified value or key will be included.

Listing 4-35. Filter

```
let ref = database.ref('products');
ref.orderByChild('price').limitToFirst(1);
```

The returned `firebase.database.Query` objects from filtering and sorting methods can be chained together to create complicated queries. There can be multiple filtering conditions, but only one sorting method is allowed.

Navigation

Given a Reference object, we can use it to navigate in the database tree. Table 4-6 shows the methods or properties for navigation. Listing 4-36 shows how to use these methods or properties.

Table 4-6. *Methods or properties for navigation*

Method	Description
<code>child(path)</code>	Get the Reference object at the specified relative path.
<code>parent</code>	Get the Reference object to its parent path.
<code>root</code>	Get the Reference object to the database's root path.

Listing 4-36. Navigation

```
let ref = database.ref('products');
ref.child('00001');
// -> path is "/products/00001"
ref.parent;
// -> path is "/"
ref.root;
// -> path is "/"
```

Hacker News API

After understanding how to interact with Firebase database in JavaScript, we can now discuss how to integrate Hacker News API (<https://github.com/HackerNews/API>) in the app. The Hacker News API is just a public Firebase database at URL <https://hacker-news.firebaseio.com/v0>.

AngularFire2

In the last section, we discussed how to use the general Firebase JavaScript SDK. For Angular 2 apps, a better choice is AngularFire2 (<https://github.com/angular/angularfire2>), which makes interacting with Firebase much easier. We use yarn to install the AngularFire2 package.

```
$ yarn add angularfire2
```

Now we need to add AngularFire2 module into the Ionic app. In Listing 4-37, the content of the `firebaseConfig` object only contains the property `databaseURL` that points to the Hacker News public database. In the `NgModule` decorator, we need to add `AngularFireModule.initializeApp(firebaseConfig)` to the array of imported modules.

Listing 4-37. AppModule with AngularFire2 config

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { AngularFireModule } from 'angularfire2';

export const firebaseConfig = {
  databaseURL: 'https://hacker-news.firebaseio.com',
};

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

After importing the AngularFire2 module, we can inject instances of AngularFire service into components that need to interact with Firebase databases; see Listing 4-38. Methods related to Firebase databases are in the namespace `database` of the AngularFire object.

Listing 4-38. Use AngularFire2 in components

```
import { Component } from '@angular/core';
import { AngularFire, FirebaseListObservable } from 'angularfire2';

@Component({
  selector: 'app-component',
  templateUrl: 'app.component.html',
})
export class AppComponent {
  items: FirebaseListObservable<any[]>;
  constructor(private af: AngularFire) {
    this.items = af.database.list('/items');
  }
}
```

To manipulate the data at given path in the Firebase database, we use methods `af.database.object()` or `af.database.list()` to create bindings. In the code below, a binding to path `/products/00001` is created. The type of the variable `product` is `FirebaseObjectObservable`.

```
let product = af.database.object('/products/00001');
```

To get the actual data, we use Angular 2 `async` pipe in the template to get the value.

```
<span>{{ (product | async)?.name }}</span>
```

`FirebaseObjectObservable` also has methods `set()`, `update()` and `remove()` to update the data, which have the same meaning as in the `Reference` object. `FirebaseObjectObservable` has two special properties: `$key` to get the key and `$value` to get the data of a primitive type.

In the code below, the return type of the method `list()` is `FirebaseListObservable`. We can use the directive `*ngFor` with the `async` pipe to display the results in the template. `FirebaseListObservable` has the method `push()` to add new values to the list. `FirebaseListObservable` also has the method `update()` and `remove()` to update the list.

```
let products = af.database.list('/products');
```

API

Now we can implement the `ItemService` using Hacker News API. Hacker News doesn't have a single API to get the list of top stories. We need to use path <https://hacker-news.firebaseio.com/v0/topstories> to get a list of ids of the top stories, then use those ids to get details of each story at path

[https://hacker-news.firebaseio.com/v0/item/\\${item_id}](https://hacker-news.firebaseio.com/v0/item/${item_id}). For example, given the story id 9893412, the URL to get its details is <https://hacker-news.firebaseio.com/v0/item/9893412>. Listing 4-39 shows the sample JSON content of a story.

Listing 4-39. Sample JSON content of a story

```
{
  "by" : "Thorondor",
  "descendants" : 134,
  "id" : 9893412,
  "kids" : [ 9894173, 9893737, ..., 9893728, 9893803 ],
  "score" : 576,
  "text" : "",
  "time" : 1436987690,
  "title" : "The Icy Mountains of Pluto",
  "type" : "story",
  "url" : "https://www.nasa.gov/image-feature/the-icy-mountains-of-pluto"
}
```

Implement ItemService

Listing 4-40 shows the implementation of `ItemService` that uses Hacker News API. `this.af.database.list('/v0/topstories')` returns a `Observable<number[]>` instance. Each value in the `Observable<number[]>` is an array of all top stories ids. Then we use the `map` operator to only select the subset of ids base on the value of parameters `offset` and `limit`. Each story id is mapped to an `Observable<Item>`. All these `Observable<Item>` objects are combined using the `combineLatest` operator to create a `Observable<Item[]>` object. The `mergeMap` operator is used to unwrap the inner `Observable<Observable<Item[]>>`, so the outer observable is still `Observable<Item[]>`. The final `map` operator creates the `Observable<Items>` from the `Observable<Item[]>` object.

Listing 4-40. ItemService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/mergeMap';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/combineLatest';
import { AngularFire } from 'angularfire2';
import { Items } from '../model/Items';

@Injectable()
export class ItemService {
  constructor(private af: AngularFire) {
```

```

    }
    load(offset: number = 0, limit: number = 10): Observable<Items> {
      return this.af.database.list('/v0/topstories')
        .map(ids => ids.slice(offset, offset + limit))
        .mergeMap((ids: any[]) => Observable.combineLatest(...(ids.map(id =>
this.af.database.object('/v0/item/' + id.$value))))))
        .map(items => ({
          offset,
          limit,
          total: limit,
          results: items,
        }));
    }
  }
}

```

Manage the Changes

The current implementation of `ItemService` uses `Observable` operators to watch for changes in the Firebase database and emit values. However, some of those updates may not be necessary. For example, when a story is updated, a new `Items` object that contains all items is emitted from the `Observable<Items>` object and triggers an update of the whole UI. This is a bad user experience. We can improve the performance by changing the data model.

Currently, we use `Item[]` in model `Items` to represent the items, which means all `Item` objects need to be resolved asynchronously before creating the `Items` object for the `ItemsComponent` to render. This makes users wait longer than expected. We can update the model `Items` to use `Observable<Item>[]` as the type of items; see Listing 4-41. Because each item is represented as its own `Observable<Item>` object, it can update itself separately.

Listing 4-41. Updated model `Items`

```

import { Observable } from 'rxjs';
import { Item } from './Item';

export interface Items {
  offset: number;
  limit: number;
  total?: number;
  results: Observable<Item>[];
}

```

After updating the model, all the test specs also need to be updated to use the new model. The important change is in the template files `items.html` of `ItemsComponent` and `item.html` of `ItemComponent`. In Listing 4-42 of the `items.html`, since the type of item is changed to `Observable<Item>`, we need to use the `async` pipe when passing items to the `ItemComponent`.

Listing 4-42. Updated items.html

```
<ion-list *ngIf="items.results.length > 0">
  <ion-item *ngFor="let item of items.results">
    <item [item]="item | async"></item>
  </ion-item>
</ion-list>
<p *ngIf="items.results.length === 0">
  No items.
</p>
```

In Listing 4-43 of the `item.html`, after the introduction of `Observable`, the property `item` of `ItemComponent` could be null, so we need to add check using `ngIf`. If the property `item` is null, then a loading message is displayed.

Listing 4-43. Updated item.html

```
<div *ngIf="!item">
  Loading...
</div>
<div *ngIf="item">
  <h2 class="title">{{ item.title }}</h2>
  <div>
    <span>
      <ion-icon name="bulb"></ion-icon>
      {{ item.score }}
    </span>
    <span>
      <ion-icon name="person"></ion-icon>
      {{ item.by }}
    </span>
    <span>
      <ion-icon name="time"></ion-icon>
      {{ item.time | timeAgo }}
    </span>
  </div>
  <div>
    <span>
      <ion-icon name="link"></ion-icon>
      {{ item.url }}
    </span>
  </div>
</div>
```

Now we can simplify the implementation of the `ItemService`; see Listing 4-44. After slicing the original array of Firebase item id object, we use a `map(v => v.$value)` operation to turn it into an array of item ids. The `distinctUntilChanged` operator is used to make sure that duplicate item ids won't be emitted to trigger unnecessary UI updates. Then the `map` operator is used to transform the array of item ids into an array of `Observable<Item>`.

Listing 4-44. Updated ItemService

```

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import * as isEqual from 'lodash.isequal';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/distinctUntilChanged';
import { AngularFire } from 'angularfire2';
import { Items } from '../model/Items';

@Injectable()
export class ItemService {
  constructor(private af: AngularFire) {

  }
  load(offset: number = 0, limit: number = 10): Observable<Items> {
    return this.af.database.list('/v0/topstories')
      .map(ids => ids.slice(offset, offset + limit).map(v => v.$value))
      .distinctUntilChanged(isEqual)
      .map((ids: any[]) => ids.map(id => this.af.database.object('/v0/item/'
+ id)))
      .map(items => ({
        offset,
        limit,
        total: limit,
        results: items,
      }));
  }
}

```

Further Improvements

When running this app, you may see the list of stories keeps updating itself. This causes some user experience issues. If the user is currently scrolling down the list of top stories and remote data is updated, then the list may be reordered to reflect the changes. This is annoying as the user's focus is lost. The list should only be refreshed explicitly by the user, for example, clicking a refresh button or pulling down the list to trigger the refresh. We also need to allow users to see more stories by implementing pagination.

To satisfy these requirements, we need to make changes to the `ItemService` again. The current implementation of the `load()` method in `ItemService` returns a new `Observable<Items>` instance when invoked. This is unnecessary because we can reuse the same `Observable<Items>` instance for results of different queries. So we introduce a new method `get()` to retrieve the `Observable<Items>` instance. In Listing 4-45, the interface `Query` represents a query object for retrieving items. The properties `offset` and

limit are used for pagination; refresh is used to indicate whether the list of story ids should be refreshed. The property queries is a RxJS Subject that acts as the communication channel between queries and result items. The method load() now emits a new Query object into the queries, which triggers the Observable<Items> to emit new Items object based on the value of the latest Query object.

Listing 4-45. Updated ItemService

```
import { Injectable } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/combineLatest';
import 'rxjs/add/operator/withLatestFrom';
import 'rxjs/add/operator/merge';
import 'rxjs/add/operator/skip';
import 'rxjs/add/operator/take';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/pluck';
import { AngularFire } from 'angularfire2';
import { Items } from '../model/Items';

export interface Query {
  refresh?: boolean;
  offset: number;
  limit: number;
}

@Injectable()
export class ItemService {
  queries: Subject<Query>;

  constructor(private af: AngularFire) {
    this.queries = new Subject<Query>();
  }

  load(query: Query) {
    this.queries.next(query);
  }

  get(): Observable<Items> {
    const rawItemIds = this.af.database.list('/v0/topstories')
      .map(ids => ids.map(v => v.$value));
    const itemIds = Observable.combineLatest(
      rawItemIds,
      this.queries,
      (ids, query) => ({ids, query}),
    ).filter(v => v.query.refresh)
      .pluck('ids');
```

```

const selector = ({offset, limit}, ids) => ({
  offset,
  limit,
  total: ids.length,
  results: ids.slice(offset, offset + limit)
    .map(id => this.af.database.object('/v0/item/' + id))
});
return Observable.merge(
  this.queries.combineLatest(itemIds, selector).take(1),
  this.queries.skip(1).withLatestFrom(itemIds, selector)
);
}
}

```

The implementation of the method `get()` is a bit complicated, but it also shows the beauty of RxJS on how to simplify state management. `rawItemIds` is an `Observable<number[]>` object that represents the story ids. The method `Observable.combineLatest()` combines values from `rawItemIds` and `queries` to create an `Observable` of item ids and `Query` object. The filter operator chained after `Observable.combineLatest()` is to make sure that values are only emitted when the property `refresh` is set to `true` in the `Query` object. Then we use `pluck` operator to extract the ids from the object. Now the `itemIds` is an `Observable` of ids that emits the latest ids only when the property `refresh` is set to `true`.

The function `selector` is to combine a `Query` object and item ids to create `Items` objects. This is done by slicing the array of item ids using `offset` and `limit` from the `Query` object, then mapping the item ids to `FirestoreObjectObservable<Item>s`. The result `Observable<Items>` is a merge of `this.queries.combineLatest(itemIds, selector).take(1)` and `this.queries.skip(1).withLatestFrom(itemIds, selector)`. The merge is required to handle the case that `Observable itemIds` may start emitting values after the `Observable queries` have emitted the initial `Query` object. In this case, a single `withLatestFrom()` doesn't work because the emitted value in `Observable itemIds` can only be picked up when the `Observable queries` emits another value. `this.queries.combineLatest(itemIds, selector).take(1)` waits until both `queries` and `itemsId` to emit at least one value, then combine the values using the `selector` function. The `take(1)` means only the first value is used. All the rest values come from the second `Observable`. `this.queries.skip(1).withLatestFrom(itemIds, selector)` uses `skip(1)` to bypass the first value in `queries`, because the first value is already captured in the first `Observable` object. For all the following values in `queries`, `withLatestFrom()` is used to combine `Query` object and item ids using the `selector` function.

Figure 4-3 shows the top stories page with a button to load more stories.

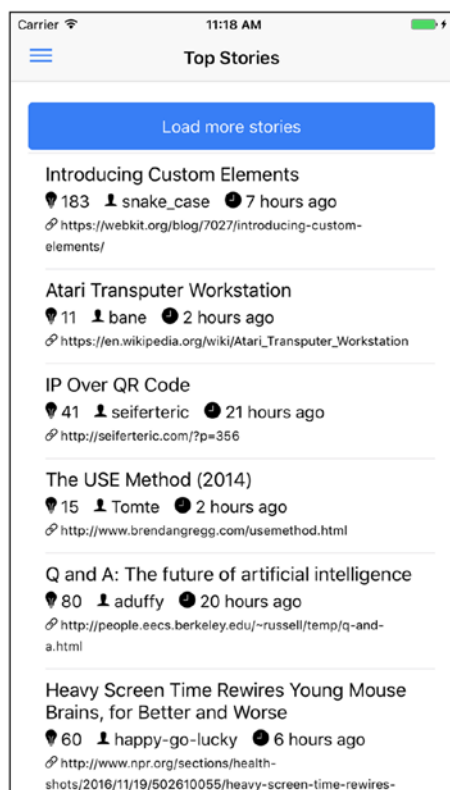


Figure 4-3. Top stories page with load button

Pagination and Refresh

Now we can add the UI for pagination and refresh.

Button

Ionic provides different styles for standard HTML buttons. Adding the directive `ion-button` to a HTML `<button>` element turns it into an Ionic button. There are three types of buttons.

- Default style – Text with filled background.
- Outline style – Text with transparent background and borders, enabled by adding the property `outline`.
- Clear style – Text with transparent background and no borders, enabled by adding the property `clear`.

To change the color of a button, use the property `color` to specify the color. Possible colors are `light`, `primary`, `secondary`, `danger` and `dark`. By default, the `primary` color is used.

For buttons with borders, we can add the property `round` to make them use round borders. For layout purpose, adding the property `block` to a button makes it take 100% of its parent's width. The property `display` is also set to `block`. To change the size of a button, adding the property `full` to a button makes it take 100% of its parent's width and remove left and right borders. Adding the property `large` or `small` makes a button larger or smaller, respectively.

Buttons can also contain icons. To add an icon to a button, we need to add a component `ion-icon` as the child of the directive `ion-button`. To control the position of the icon in the button, we can use the property `icon-left` or `icon-right` in `ion-button`.

Pagination

We add the HTML markup to show the buttons in the `top-stories.html` file; see Listing 4-46.

Listing 4-46. Add pagination buttons

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Top Stories</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <span class="buttons">
    <button ion-button icon-left color="light" [disabled]="!hasPrevious()"
(click)="previous()">
      <ion-icon name="arrow-back"></ion-icon>
      Prev
    </button>
    <button ion-button [disabled]="!canRefresh()" (click)="refresh()">
      <ion-icon name="refresh"></ion-icon>
    </button>
    <button ion-button icon-right color="light" [disabled]="!hasNext()"
(click)="next()">
      <ion-icon name="arrow-forward"></ion-icon>
      Next
    </button>
  </span>
</ion-content>
```

```
</span>
<hn-items *ngIf="items" [items]="items"></hn-items>
</ion-content>
```

The `top-stories.ts` file is also updated to use the new methods of `ItemService` and include action handlers for pagination and refresh buttons; see Listing 4-47. The value of the property `offset` is the index of the first loaded item in the whole list. We use `offset` to determine if the previous or next button should be enabled. Pagination is done by updating the value of `offset`.

Listing 4-47. Updated TopStories

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { NavController } from 'ionic-angular';
import { Subscription } from "rxjs";
import { Items } from '../model/Items';
import { ItemService } from '../services/ItemService';

@Component({
  selector: 'page-top-stories',
  templateUrl: 'top-stories.html'
})
export class TopStories implements OnInit, OnDestroy {
  items: Items;
  subscription: Subscription;
  offset: number = 0;
  limit: number = 10;
  constructor(public navCtrl: NavController, private itemService:
ItemService) {}

  ngOnInit(): void {
    this.subscription = this.itemService.get().subscribe(items => this.items
    = items);
    this.doLoad(true);
  }

  ngOnDestroy(): void {
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
  }

  hasPrevious(): boolean {
    return this.offset > 0;
  }

  previous(): void {
    if (!this.hasPrevious()) {
      return;
    }
  }
}
```

```
    }
    this.offset -= this.limit;
    this.doLoad(false);
}

hasNext(): boolean {
    return this.items != null && (this.offset + this.limit) < this.items.
        total;
}

next(): void {
    if (!this.hasNext()) {
        return;
    }
    this.offset += this.limit;
    this.doLoad(false);
}

canRefresh(): boolean {
    return this.items != null;
}

refresh() : void {
    if (!this.canRefresh()) {
        return;
    }
    this.offset = 0;
    this.doLoad(true);
}

doLoad(refresh: boolean): void {
    this.itemService.load({
        offset: this.offset,
        limit: this.limit,
        refresh,
    });
}
}
```

Figure 4-4 shows the top stories page with buttons for pagination and refresh.

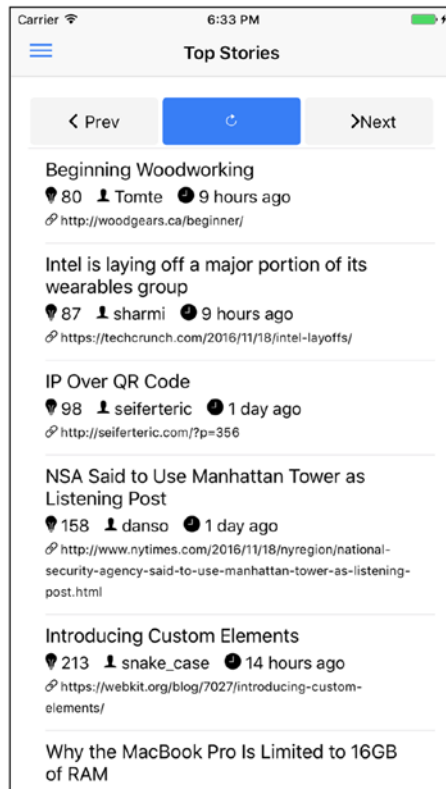


Figure 4-4. Top stories page with pagination and refresh

Advanced List

So far we implemented the top stories list with basic pagination and refresh support. But the current implementation is more like a traditional web app instead of a mobile app. Mobile apps have their own UI patterns. For pagination and refresh, it's better to use infinite scrolling and pull-to-refresh.

If infinite scrolling is enabled on the list, when the user scrolls down the list and nearly hits the bottom of the page, an action is triggered to load items of the next page. The user can continuously read through the list without clicking any buttons. Ionic provides built-in support for infinite scrolling with components `ion-infinite-scroll` and `ion-infinite-scroll-content`.

Pull-to-refresh is a very common feature in mobile apps. When displaying a list of items, a user can pull down the list and release, and then the list will refresh to get the latest data. Ionic also provides built-in support for this feature with components `ion-refresher` and `ion-refresher-content`.

The component `ion-infinite-scroll` and `ion-refresher` follow the similar usage pattern. After adding these two components into the page, they invoke the provided callback functions when certain events happen, then wait for client code to notify them when the processing is finished. The components `ion-infinite-scroll-content` and `ion-refresher-content` are used to customize the feedback UI when the processing is still underway.

In Listing 4-48, `ion-refresher` is added to the top of the list of items, while `ion-infinite-scroll` is added to the bottom. For `ion-refresher`, we add a handler for the `ionRefresh` event and invoke the method `refresh()` with the current event object `$event` as the argument. The event object `$event` is the instance of the current `ion-refresher` component. The property `enabled` controls if the refresher is enabled or not. For the component `ion-infinite-scroll`, the event `ionInfinite` is handled using the method `load()`.

Listing 4-48. Add `ion-refresher` and `ion-infinite-scroll`

```
<ion-content padding>
  <ion-refresher [enabled]="canRefresh()" (ionRefresh)="refresh($event)">
    <ion-refresher-content></ion-refresher-content>
  </ion-refresher>
  <hn-items *ngIf="items" [items]="items"></hn-items>
  <ion-infinite-scroll [enabled]="hasNext()" (ionInfinite)="load($event)">
    <ion-infinite-scroll-content></ion-infinite-scroll-content>
  </ion-infinite-scroll>
</ion-content>
```

Listing 4-49 is the updated code of `top-stories.ts` to support infinite scrolling and pull-to-refresh. In the method `ngOnInit()`, when the `Observable<Items>` emits a new value, we need to check if it's a refresh request or normal loading request. For refresh requests, we simply assign the value `Items` to `this.items`. For a normal loading request, we need to merge the items from the value `Items` with those items already contained in `this.items`, so new items are displayed at the bottom of the old items. The property `refresh` is newly added to model `Items` with the value that comes from the property `refresh` in the `Query` object.

In the method `load()`, the instance of current `ion-infinite-scroll` component is saved to `this.infiniteScrollComponent`. When the loading of items is completed, the method `complete()` of `infiniteScrollComponent` must be called to stop the loading spinner. That's the reason to have method `notifyScrollComplete()` called in the subscription logic of `Observable<Items>`. The method `refresh()` also has the same logic to save the instance of the current `ion-refresher` component. We also call `notifyRefreshComplete()` method to stop the spinner for refresh queries.

Listing 4-49. Updated TopStories

```
import { Component, OnInit } from '@angular/core';
import { NavController } from 'ionic-angular';
import * as concat from 'lodash.concat';
import { Items } from '../model/Items';
import { ItemService } from '../services/ItemService';

@Component({
  selector: 'page-top-stories',
  templateUrl: 'top-stories.html'
})
export class TopStories implements OnInit {
  items: Items;
  offset: number = 0;
  limit: number = 10;
  infiniteScrollComponent: any;
  refresherComponent: any;
  constructor(public navCtrl: NavController, private itemService:
ItemService) {}

  ngOnInit(): void {
    this.itemService.get().subscribe(items => {
      if (items.refresh) {
        this.items = items;
        this.notifyRefreshComplete();
      } else {
        this.items.results = concat(this.items.results, items.results);
        this.notifyScrollComplete();
      }
    });
    this.doLoad(true);
  }

  hasNext(): boolean {
    return this.items != null && this.offset < this.items.total;
  }

  next(): void {
    this.offset += this.limit;
    this.doLoad(false);
  }

  canRefresh(): boolean {
    return this.items != null;
  }

  refresh(refresher): void {
    this.refresherComponent = refresher;
  }
}
```

```

        if (this.canRefresh()) {
            this.doRefresh();
        }
    }

    load(infiniteScroll) {
        this.infiniteScrollComponent = infiniteScroll;
        if (this.hasNext()) {
            this.next();
        }
    }

    doRefresh() : void {
        this.offset = 0;
        this.doLoad(true);
    }

    doLoad(refresh: boolean): void {
        this.itemService.load({
            offset: this.offset,
            limit: this.limit,
            refresh,
        });
    }

    private notifyScrollComplete(): void {
        if (this.infiniteScrollComponent) {
            this.infiniteScrollComponent.complete();
        }
    }

    private notifyRefreshComplete(): void {
        if (this.refresherComponent) {
            this.refresherComponent.complete();
        }
    }
}

```

Customization

We can also customize the behaviors of the components `ion-infinite-scroll` and `ion-refresher`.

ion-infinite-scroll

By default, Ionic shows a platform specific spinner when the infinite scrolling action is still in progress. We can customize the spinner and text displayed using the component `ion-infinite-scroll-content`. `ion-infinite-scroll-content` has two properties: `loadingSpinner` to set the spinner style, possible values are `ios`, `ios-small`, `bubbles`, `circles`, `crescent`, and `dots`; `loadingText` to set the text to display. In Listing 4-50, we configure the `ion-infinite-scroll` to use a different spinner and text.

Listing 4-50. ion-infinite-scroll customization

```
<ion-infinite-scroll-content
  loadingSpinner="circles"
  loadingText="Loading...">
</ion-infinite-scroll-content>
```

The `ion-infinite-scroll` also has a property `threshold` to control when to trigger the `ionInfinite` events. The value of property `threshold` can be a percentage or absolute pixel value. When the current scrolling position reaches a specified threshold, the `ionInfinite` event is triggered. The default value of the threshold is 15%. It's recommended to provide a relatively large value for threshold so that the loading operation can have enough time to finish to provide the user with a smooth reading experience.

ion-refresher

We can use the property `pullMin` of `ion-refresher` to control the distance that the user needs to pull down before the `ionRefresh` event can be triggered. If the user keeps pulling down and reaches the distance specified in the property `pullMax`, the `ionRefresh` event is triggered automatically even though the user doesn't release yet. The default value of `pullMin` and `pullMax` is 60 and `pullMin + 60`, respectively.

Like `ion-infinite-scroll-content`, the component `ion-refresher-content` also provides several properties to control the style of `ion-refresher`.

Listing 4-51 shows an example of customizing the `ion-refresher`.

- `pullingIcon` – Ionic icon to display when the user begins to pull down.
- `pullingText` – Text to display when the user begins to pull down.
- `refreshingSpinner` – A spinner to display when refreshing. Possible values are the same as the property `loadingSpinner` of `ion-infinite-scroll-content`.
- `refreshingText` – Text to display when refreshing.

Listing 4-51. ion-refresher customization

```
<ion-refresher-content
  pullingIcon="arrow-dropdown"
  pullingText="Pull to refresh"
  refreshingSpinner="bubbles"
  refreshingText="Loading...">
</ion-refresher-content>
```

Testing

As usual, we need to add test specs for infinite scrolling and pull-to-refresh features. In Listing 4-52, two new specs are added to existing `top-stories.spec.ts`. In the first test spec, the component is the instance of `TopStoriesPage`. We invoke `component.next()` directly to simulate the items loading triggered by scrolling. We check that there should be 20 items displayed. In the second test spec, we use `component.next()` to load more items, then use `component.doRefresh()` to simulate the refreshing event. We check that the number of items should be 10 after the refresh.

Listing 4-52. Test for scrolling and refresh

```
let fixture: ComponentFixture<TopStoriesPage> = null;
let component: any = null;

describe('top stories page', () => {

  it('should show more items when scrolling down', async(() => {
    fixture.detectChanges();
    fixture.whenStable().then(() => {
      fixture.detectChanges();
      component.next();
      fixture.detectChanges();
      fixture.whenStable().then(() => {
        let debugElements = fixture.debugElement.queryAll(By.css('h2'));
        expect(debugElements.length).toBe(20);
        expect(debugElements[10].nativeElement.textContent).toContain('Item 11');
      });
    });
  }));

  it('should show first 10 items when refresh', async(() => {
    fixture.detectChanges();
    fixture.whenStable().then(() => {
      fixture.detectChanges();
      component.next();
      fixture.detectChanges();
    });
  }));
});
```

```
fixture.whenStable().then(() => {
    let debugElements = fixture.debugElement.queryAll(By.css('h2'));
    expect(debugElements.length).toBe(20);
    expect(debugElements[10].nativeElement.textContent).toContain('Item 11');
    component.doRefresh();
    fixture.detectChanges();
    fixture.whenStable().then(() => {
        let debugElements = fixture.debugElement.queryAll(By.css('h2'));
        expect(debugElements.length).toBe(10);
        expect(debugElements[0].nativeElement.textContent).toContain('Item 1');
    });
});
});
});
});
});
```

Loading and Error

All the code so far only deals with successful scenarios. An app should be robust and be ready to handle all kinds of expected or unexpected errors. For the app to be more responsive and user friendly, we also need to add loading indicators when the app is performing certain actions that require server interactions.

Loading

When the user triggers the refresh of the top stories page, a loading indicator should be displayed. Ionic provides a built-in loading component that renders spinner and text in a modal dialog.

In the `ItemService`, when a new `Query` object is received, we should immediately emit an `Items` object to represent the loading event and trigger the display of the loading indicator. When the items loading is finished, the subsequent `Items` object is emitted and triggers the dismissal of the loading indicator. We add a new `Boolean` property `loading` to the model `Items`.

In the updated implementation of `ItemService` of Listing 4-53, we use another `Observable<Items>` instance `loadings` to emit all loading events. The `Items` objects in the `loadings` observable all have property `loading` set to `true`. They are emitted when the subject queries receives `Query` objects with the property `refresh` set to `true`. The returned `Observable<Items>` object of the method `ItemService.get()` is the merge of the `loadings` observable and existing `items` observable.

Listing 4-53. Updated ItemService

```

@Injectable()
export class ItemService {

  get(): Observable<Items> {
    // code omitted
    const loadings = this.queries
      .filter(v => v.refresh)
      .map(query => ({
        loading: true,
        offset: query.offset,
        limit: query.limit,
        results: [],
      }));
    const items = Observable.merge(
      this.queries.combineLatest(itemIds, selector).take(1),
      this.queries.skip(1).withLatestFrom(itemIds, selector)
    );
    return Observable.merge(loadings, items);
  }
}

```

LoadingController and Loading

In the `top-stories.ts` file of Listing 4-54, we use `LoadingController` and `Loading` from Ionic to show the loading indicator. `LoadingController` is responsible for creating new instances of `Loading` components. The instance of `LoadingController` is injected into the component `TopStoriesPage`. In the method `showLoading()`, a new instance of the `Loading` component is created using `LoadingController.create()`, then it's displayed using `Loading.present()`. In the method `hideLoading()`, `Loading.dismiss()` is used to dismiss the loading indicator. Once a `Loading` component is dismissed, it cannot be used anymore, that is, you cannot make it display again. A new instance needs to be created every time.

In the subscription logic of `Observable<Items>`, if the current `Items` object represents a loading event, we just show the loading indicator and return. Otherwise, the loading indicator is dismissed and items are rendered.

Listing 4-54. Updated TopStories

```

import { LoadingController, Loading } from 'ionic-angular';

@Component({
  selector: 'page-top-stories',
  templateUrl: 'top-stories.html'
})

```

```
export class TopStories implements OnInit {
  loading: Loading;
  constructor(public navCtrl: NavController,
               private itemService: ItemService,
               public loadingCtrl: LoadingController) {}

  ngOnInit(): void {
    this.itemService.get().subscribe(items => {
      if (items.loading) {
        this.showLoading();
        return;
      }
      this.hideLoading();
      // code omitted
    });
    this.doLoad(true);
  }

  // code emitted

  private showLoading(): void {
    this.hideLoading();
    this.loading = this.loadingCtrl.create({
      content: 'Loading...',
    });
    this.loading.present();
  }

  private hideLoading(): void {
    if (this.loading) {
      this.loading.dismiss();
    }
  }
}
```

In the current implementation, when a refresh query is issued, the loading indicator is always displayed. In fact, except for the first time loading, all the other refresh requests finish immediately. The first loading request needs to wait until the top stories ids to be retrieved. But all the subsequent refresh requests just use the latest retrieved top stories ids, even though these ids may not be up to date. We can only show the loading indicator for the first time.

We just need to change loadings to emit only one single value; see Listing [4-55](#).

Listing 4-55. Emit only one loading event

```
const loadings = Observable.of({
  loading: true,
  offset: 0,
  limit: 0,
  results: [],
});
```

Testing

To verify that the loading indicators are displayed and dismissed as expected, we need to add test specs for this scenario. The test spec is different than previous specs as it uses spies from Jasmine.

At first, we need to update the method `get()` of `ItemServiceMock` to emit the loading event. This can be done easily with RxJS `startWith` operator; see [Listing 4-56](#).

Listing 4-56. ItemServiceMock to emit loading event

```
@Injectable()
export class ItemServiceMock extends ItemService {
  get(): Observable<Items> {
    return this.queries.map(query => ({
      refresh: query.refresh,
      offset: query.offset,
      limit: query.limit,
      total: query.offset + query.limit + 1,
      results: generateItems(query.offset, query.limit),
    } as Items)).startWith({
      loading: true,
      offset: 0,
      limit: 0,
      results: [],
    });
  }
}
```

To verify that a Loading component is created by the LoadingController and displayed, we need to use test doubles for both components Loading and LoadingController. As in [Listing 4-57](#), `loadingStub` is the spy object created using `jasmine.createSpyObj()`. Both methods `present` and `dismiss` are spies. `loadingControllerStub` is the spy object for LoadingController that has the method `create()` to always return the `loadingStub` object.

Listing 4-57. Create Jasmine spy objects

```
const loadingStub = jasmine.createSpyObj('loading', ['present', 'dismiss']);
const loadingControllerStub = {
  create: jasmine.createSpy('create').and.returnValue(loadingStub),
};
```

These two test doubles are declared as value providers when invoking the method `TestUtils.beforeEachCompiler()`. The property `useValue` means all the instances of `LoadingController` use the same `loadingControllerStub` object. During the testing, the `TopStoriesPage` class uses `loadingControllerStub.create()` to create a new `Loading` component and get the spy object `loadingStub` as the result. All the calls to methods `present` and `dismiss` are captured by the `loadingStub` object and this information is available for verifications.

```
{provide: LoadingController, useValue: loadingControllerStub}
```

In Listing 4-58, we first get the Angular 2 injector using `fixture.debugElement.injector`, then get the `loadingControllerStub` object using the injector by looking up the token `LoadingController`. We can verify that the method `loadingControllerStub.create()` has been called once to create the loading component. For the created loading component, methods `present` and `dismiss` should also be called once.

Listing 4-58. Test spec for loading

```
it('should display loading indicators', async(() => {
  const loadingController = fixture.debugElement.injector.
    get>LoadingController);
  fixture.detectChanges();
  expect(loadingController.create).toHaveBeenCalledTimes(1);
  expect(loadingController.create().present).toHaveBeenCalledTimes(1);
  expect(loadingController.create().dismiss).toHaveBeenCalledTimes(1);
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    let debugElements = fixture.debugElement.queryAll(By.css('h2'));
    expect(debugElements.length).toBe(10);
  });
}));
```

Error Handling

To handle errors in the `Observables`, we can add an error handler in the subscriber. For the `Observable<Items>` of top stories ids, when an error occurred, the observable object is terminated. We need to create a new `Observable<Items>` object to retry the loading. We can move the code in the

method `ngOnInit()` into a separate method `init()` and invoke this method upon errors; see Listing 4-59.

Listing 4-59. Error handling

```
export class TopStoriesPage implements OnInit {
  constructor(public toastCtrl: ToastController) {}

  ngOnInit(): void {
    this.init();
  }

  init(): void {
    this.itemService.get().subscribe(items => {

    }, error => {
      this.showError();
      this.init();
    });
    this.doLoad(true);
  }
}
```

Toast

Ionic has a built-in component to show toast notifications. We can use `ToastController` to create new toast notifications. The instance of `ToastController` is injected into `TopStories` as the property `toastCtrl`. The method `showError()` in Listing 4-60 creates a `Toast` object and shows it.

Listing 4-60. Show error toast notifications

```
private showError(): void {
  const toast = this.toastCtrl.create({
    message: 'Failed to load items, retry now...',
    duration: 3000,
    showCloseButton: true,
  });
  toast.present();
}
```

When creating a new toast notification using the method `create()`, we can pass an object to configure the toast notification. The following configuration properties are supported.

- `message` – Message to display.
- `duration` – Number of milliseconds before the toast is dismissed.

- `position` – Position to show the toast; possible values are top, middle and bottom.
- `showCloseButton` – Whether to show a button to close the toast.
- `closeButtonText` – Text of the close button.
- `dismissOnPageChange` – Whether to dismiss the toast when navigating to a different page.
- `cssClass` – Extra CSS classes to add to the toast.

The created Toast object also has methods `present()` and `dismiss()` to show and dismiss it. When the toast is created with the property `duration` set, then it's not required to call the method `dismiss()` explicitly. The toast dismisses itself after the configured duration time elapsed.

Testing

To test the error handling, we need the `ItemService` to generate errors. We use a Jasmine spy to control the return values of the method `get()` of `ItemService` using Jasmine's method `returnValues()`; see Listing 4-61. The first return value is an `Observable` object that only throws an error. The second return value is a normal `Observable` object with items. When the test runs, the first request to load items will fail, so we verify that the error toast notification is displayed. Then we verify that the second loading is triggered automatically and renders the items correctly.

Listing 4-61. Test spec for error handling

```
it('should handle errors', async(() => {
  const itemService = fixture.debugElement.injector.get(ItemService);
  spyOn(itemService, 'get')
    .and.returnValues(
      Observable.throw(new Error('boom!')),
      Observable.of({
        refresh: true,
        offset: 0,
        limit: 10,
        results: generateItems(0, 10),
      })
    );
  const toastController = fixture.debugElement.injector
    .get(ToastController);
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(toastController.create).toHaveBeenCalledTimes(1);
  });
});
```

```

    expect(toastController.create().present).toHaveBeenCalledTimes(1);
    let debugElements = fixture.debugElement.queryAll(By.css('h2'));
    expect(debugElements.length).toBe(10);
  });
});

```

Figure 4-5 shows the screenshot of the final version of top stories page.

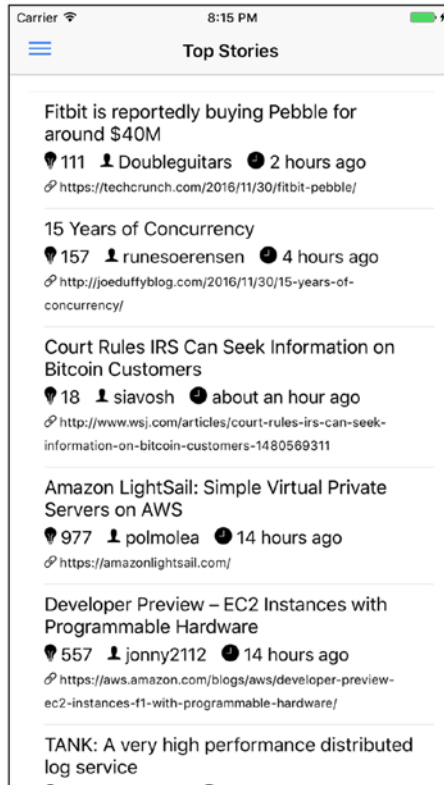


Figure 4-5. Top stories page with infinite scrolling and pull-to-refresh

Summary

This longest chapter lays the groundwork for implementing various user stories in the app. It shows how to organize a user story implementation as components and services and uses Jasmine and Karma to test them. Firebase basics and API are also covered. When using the Ionic 2 list component to display top stories, we also demonstrate the advanced features like infinite scrolling and pull-to-refresh. Loading indicators and error handling are also added. After this chapter, you should know how to implement a well-organized and tested user story. In the next chapter, we'll implement the user story to view stories.

Chapter 5

View Story

After we finished the `TopStoriesPage` to list top stories, we are going to allow users to view the actual web pages of stories. The basic solution is quite simple. We just need to use the standard HTML `<a>` elements in the item's template. As in Listing 5-1, we added the `<a>` element with the `href` attribute set to `item.url`. The old `div` element that contains the url is removed.

Listing 5-1. Use `<a>` for a story's URL

```
<h2 class="title">
  <a [href]="item.url">{{ item.title }}</a>
</h2>
```

When we run the code inside of an emulator or a device, clicking the title opens the platform default browser and shows the web page. On Android, the user can use the back button to return to the app; while on the iOS platform, the user can use the back button on the top left of the nav bar to return to the app.

In App Browser

Using `<a>` elements is a valid solution for opening links in the app, but it has one major disadvantage that the user needs to leave the app to read the stories and return to the app after that. This creates a bad user experience for interrupting the user's normal flow. Another disadvantage is that we cannot customize the behaviors of the opened browser windows. A better solution is to use the Cordova inappbrowser plugin (<https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-inappbrowser/>).

To use Cordova plugins in Ionic apps, Ionic creates a library Ionic Native (<https://ionicframework.com/docs/v2/native/>) as a wrapper for Cordova

plugins. The package `ionic-native` is already installed when the app is created from the starter template.

Installation

Cordova plugins can be installed using either the command `ionic plugin add` or `cordova plugin add`. The plugins files are installed into the directory `plugins`.

```
$ ionic plugin add cordova-plugin-inappbrowser --save
```

After the `inappbrowser` plugin is installed in the app, we can access its API using the `InAppBrowser` module.

Open a URL

To open the story's web page, we can create a new instance of `InAppBrowser` or use the static method `InAppBrowser.open(url, target, options)`. The constructor of `InAppBrowser` has the same parameters as the method `open()`. `url` is the URL to open. `target` is the target to load the URL with following possible values:

- `_self` – If the URL is in the white list, opens it in the `WebView`; otherwise it opens in the `InAppBrowser`.
- `_blank` – Opens in the `InAppBrowser`.
- `_system` – Opens in the system web browser.

The parameter `options` is a string to configure the on/off status for different features in `InAppBrowser`. Different platforms have their own supported options. `location` is the only option that is supported on all platforms. The `location` option can be set to `yes` or `no` to show or hide the browser's location bar. The default value of `options` is `location=yes`. Options for different features are separated by commas, for example, `location=no,hidden=yes`.

Some common options are these:

- `hidden` – When set to `yes`, the browser is created and loads the page, but it's hidden. We need to use `InAppBrowser's show()` method to show it.
- `clearcache` – When set to `yes`, the browser's cookie cache is cleared before opening the page.
- `clearsessioncache` – When set to `yes`, the browser's session cookie cache is cleared before opening the page.

In Listing 5-2, we add a new method `openPage()` in the `ItemComponent` to open a given URL. The URL is opened using target `_blank`. Creating a new `InAppBrowser` opens the URL and shows it by default.

Listing 5-2. Open page

```
import { Component, Input } from '@angular/core';
import { Item } from '../../model/Item';
import { InAppBrowser } from 'ionic-native';

@Component({
  selector: 'item',
  templateUrl: 'item.html',
})
export class ItemComponent {
  @Input() item: Item;
  openPage(url: string): void {
    new InAppBrowser(url, '_blank');
  }
}
```

It's not a good practice to put the actual page opening logic in the `ItemComponent`. As we mentioned before, we should encapsulate this kind of logic in services. The `OpenPageService` in Listing 5-3 uses an instance property `browser` to manage the current `InAppBrowser` object. We only allow at most one `InAppBrowser` object to be opened at the same time, so the existing browser is closed before opening a new page.

Listing 5-3. OpenPageService

```
import { Injectable } from '@angular/core';
import { InAppBrowser } from 'ionic-native';

@Injectable()
export class OpenPageService {
  browser: InAppBrowser;
  open(url: string): void {
    if (this.browser) {
      this.browser.close();
      this.browser = null;
    }
    this.browser = new InAppBrowser(url, '_blank');
  }
}
```

Now we need to pass the `OpenPageService` instance to the `ItemComponent`. The instance of `OpenPageService` is injected into `TopStoriesPage` class and can be passed to the `ItemsComponent` and then to the `ItemComponent`

using the Input decorator. Passing the instance of `OpenPageService` is a straightforward and valid solution, but it's not the ideal solution because it creates unnecessary coupling between services and UI components. All the `ItemComponent` needs is a function to open the URL, so we should pass the actual function instead of the whole service object. By doing this, we minimized the coupling between services and UI components, which makes testing and maintenance easier.

The interface `OpenPage` in Listing 5-4 is the function type that represents the function required by the `ItemsComponent`.

Listing 5-4. OpenPage interface

```
export interface OpenPage {  
  (url: string): void;  
}
```

In the constructor of the `TopStoriesPage`, we set the value of the property `openPage` to the `open()` method of the `OpenPageService` instance; see Listing 5-5. `bind()` is used to make sure this keyword refers to the correct `openPageService` object when the method `open()` is invoked.

Listing 5-5. Set openPage

```
this.openPage = openPageService.open.bind(openPageService);
```

Events

`InAppBrowser` has different events to listen on.

- `loadstart` – Fired when the `InAppBrowser` starts to load a URL.
- `loadstop` – Fired when the `InAppBrowser` finishes loading a URL.
- `loaderror` – Fired when an error occurred when the `InAppBrowser` is loading a URL.
- `exit` – Fired when the `InAppBrowser` is closed.

To add listeners on a certain event, we can use the method `on(event)` of `InAppBrowser` object. The method `on()` returns an `Observable` object that emits event objects when subscribed. Unsubscribing the `Observable` removes the event listener.

With these events, we can have fine-grained control over the web page loading in the `InAppBrowser` to improve user experiences. For example, we can make the `InAppBrowser` hidden when it's loading the page and show it after the loading is completed.

Alerts

We can use components `LoadingController` and `Loading` to show a message when the page is still loading. The component `Loading` only supports adding spinners and texts, but we need to add a cancel button in the loading indicator to allow users to abort the operation. In this case, we should use Ionic components `AlertController` and `Alert`.

The usage of components `AlertController` and `Alert` is like `LoadingController` and `Loading`. `AlertController` has the same method `create()` to create `Alert` components, but the options are different. `Alert` has the same methods `present()` and `dismiss()` to show and hide the alert.

The method `create()` supports following options.

- `title` – The title of the alert.
- `subTitle` – The subtitle of the alert.
- `message` – The message to display in the alert.
- `cssClass` – Space-separated extra CSS classes.
- `inputs` – An array of inputs.
- `buttons` – An array of buttons.
- `enableBackdropDismiss` – Whether the alert should be dismissed when the user taps on the backdrop.

Alerts can include different kinds of inputs to gather data from the user. Supported inputs are text boxes, radio buttons, and check boxes. Inputs can be added by specifying the `inputs` options in the method `create()`, or using the method `addInput()` of the `Alert` object. The following properties can be used to configure an input.

- `type` – The type of the input. Possible values are `text`, `radio`, `checkbox`, `tel`, `number` and other valid HTML5 input types.
- `name` – The name of the input.
- `placeholder` – The placeholder of the input.
- `label` – The label of the input, for radio buttons and check boxes.
- `value` – The value of the input.
- `checked` – Whether the input is checked or not.
- `id` – The id of the input.

Buttons can also be added using the buttons options or the method `addButton()` of the `Alert` object. Buttons can have the following properties.

- `text` – The text of the button.
- `handler` – The expression to evaluate when the button is pressed.
- `cssClass` – Space-separated extra CSS classes.
- `role` – The role of the button; it can be `null` or `cancel`.
When `enableBackdropDismiss` is set to `true` and the user taps on the backdrop to dismiss the alert, then the handler of the button with `cancel` role is invoked.

A Better Solution

We can now create a better implementation that combines the `InAppBrowser` events and alerts to give users more feedback and control over opening web pages. In Listing 5-6, the options when creating a new `InAppBrowser` instance is `location=no,hidden=yes`, which means the location bar is hidden and the browser window is initially hidden. We subscribe to all four different kinds of events: `loadstart`, `loadstop`, `loaderror` and `exit`. In the handler of the `loadstart` event, an `Alert` is created and presented with the button to cancel the loading. In the handler of the `loadstop` event, we hide the alert and use the method `show()` of `InAppBrowser` to show the browser window. In the handler of the `loaderror` event, we create a toast notification and show it. In the handler of the `exit` event, we unsubscribe all subscriptions to the event observables and use the method `close()` of `InAppBrowser` to close the browser window.

The handler of the cancel button is the method `cancel()` of `OpenPageService`, so when the cancel button is tapped, the loading alert is dismissed and the browser window is closed.

The `RxJS Subscription` class has the method `add()` to add child subscriptions. These child subscriptions are unsubscribed when the parent subscription is unsubscribed. We can use only one `Subscription` object to manage subscriptions to four different events.

Listing 5-6. Updated OpenPageService

```

import { Injectable } from '@angular/core';
import { InAppBrowser, InAppBrowserEvent } from 'ionic-native';
import { AlertController, Alert } from 'ionic-angular';
import { ToastController } from 'ionic-angular';
import { Subscription } from "rxjs";

@Injectable()
export class OpenPageService {
  browser: InAppBrowser;
  loading: Alert;
  subscription: Subscription;
  constructor(private alertCtrl: AlertController,
               private toastCtrl: ToastController) {

  }
  open(url: string): void {
    this.cancel();
    this.browser = new InAppBrowser(url, '_blank',
    'location=no,hidden=yes');
    this.subscription = this.browser.on('loadstart').subscribe(event =>
this.showLoading());
    this.subscription.add(this.browser.on('loadstop').subscribe(event => {
      this.hideLoading();
      this.browser.show();
    }));
    this.subscription.add(this.browser.on('loaderror').subscribe(event =>
this.handleError(event)));
    this.subscription.add(this.browser.on('exit').subscribe(event => this.
cleanup()));
  }

  showLoading(): void {
    this.hideLoading();
    this.loading = this.alertCtrl.create({
      title: 'Opening...',
      message: 'The page is loading. You can press the Cancel button to stop it.',
      enableBackdropDismiss: false,
      buttons: [
        {
          text: 'Cancel',
          handler: this.cancel.bind(this),
        }
      ],
    });
    this.loading.present();
  }
}

```

```
cancel(): void {
    this.hideLoading();
    this.cleanup();
}

handleError(event: InAppBrowserEvent): void {
    this.showError(event);
    this.cleanup();
}

hideLoading(): void {
    if (this.loading) {
        this.loading.dismiss();
        this.loading = null;
    }
}

showError(event: InAppBrowserEvent): void {
    this.hideLoading();
    const toast = this.toastCtrl.create({
        message: `Failed to load the page. Code: ${event.code}, Message:
${event.message}`,
        duration: 3000,
        showCloseButton: true,
    });
    toast.present();
}

cleanup() : void {
    if (this.subscription) {
        this.subscription.unsubscribe();
        this.subscription = null;
    }
    if (this.browser) {
        this.browser.close();
        this.browser = null;
    }
}
```

The new implementation gives users a better experience when viewing web pages. Figure 5-1 shows the alert when a page is opening.

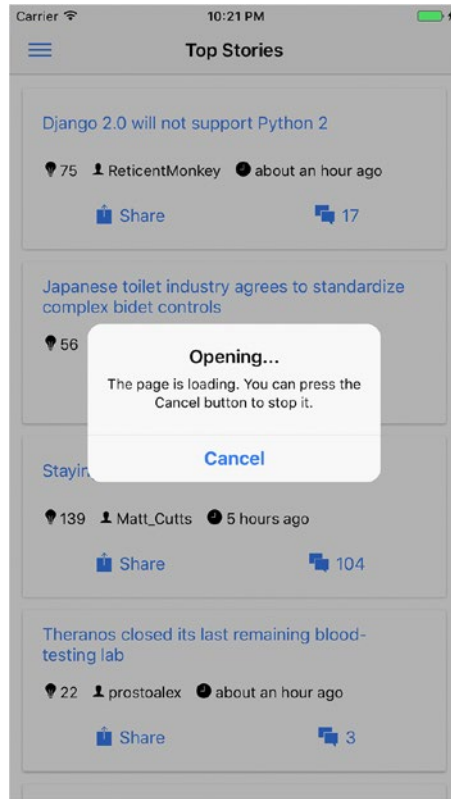


Figure 5-1. Opening page

Styles

After we removed the `<a>` elements from the template and use `<h2>` elements to represent the titles, the color of the titles is changed to black. We still want the title color to be different. Here we can use the primary color from Ionic as the text color.

In the Sass code of Listing 5-7, we use the function `color` from Ionic to get the named color `primary` from the default color map `$colors`, which is defined in the `src/theme/variables.scss` file.

Listing 5-7. Styles

```
item {
  .title {
    color: color($colors, primary);
  }
}
```

Testing

We need to add test spec for `OpenPageService`. We create a test stub object `openPageServiceStub` using a Jasmine spy, then register the stub object as the provider of `OpenPageService`.

```
const openPageServiceStub = jasmine.createSpyObj('openPage', ['open']);
```

In Listing 5-8, we get the `debugElement` of the `h2` element in the first item and use the method `triggerEventHandler()` to trigger the `click` event, which simulates the user's clicking of the `h2` element. We then verify the method `open()` of `openPageServiceStub` to have been called with the URL of the first item.

Listing 5-8. Open page test spec

```
it('should open web pages', async(() => {
  const openPageService = fixture.debugElement.injector.
    get(OpenPageService);
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    let debugElements = fixture.debugElement.queryAll(By.css('h2'));

    expect(debugElements.length).toBe(10);
    expect(debugElements[0].nativeElement.textContent).toContain('Item 1');
    debugElements[0].triggerEventHandler('click', null);
    expect(openPageService.open)
      .toHaveBeenCalledWith('http://www.example.com/item0');
  });
}));
```

Summary

In this chapter, we implement the user story to view stories. We use the plugin `InAppBrowser` to allow users to view the web pages inside of the app itself. Events of `InAppBrowser` are used to control its behavior. By using `Ionic 2 Alerts`, we can create a smooth experience for users to view web pages. In the next chapter, we'll implement the user story to view comments of stories.

Chapter 6

View Comments

So far we focused on the top stories page, which is also the index page of the app. Now we need to add new pages to show comments of a story and replies to a comment. Before we begin adding those pages, we need to talk about page navigation in Ionic.

Navigation

In the constructor of `TopStoriesPage` class, an instance of `NavController` is injected. `NavController` is used to navigate between different pages in the app. `NavController` maintains an array of pages that represent the navigation history. This array can be directly manipulated by adding or removing pages at arbitrary locations of the array. Or this array can be treated as a stack and pages can be pushed onto the stack or popped from it. The last page in the array is the currently active page.

Basic Usage

In the `app.html` of the app's root component, we use the component `ion-nav` to enable page navigation. The property `root` of `ion-nav` sets the root page of the navigation.

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

Page Navigation

Page navigation is done by using methods of `NavController` class to change the last page in the array of navigation history.

- `push(page, params, opts)` – Push a new page onto the navigation stack. `params` is the parameters passed to the new page. `opts` is options object to configure the view transition.
- `insert(insertIndex, page, params, opts)` – Insert a page into the navigation stack at a specified index.
- `insertPages(insertIndex, insertPages, opts)` – Insert an array of pages into the navigation stack at a specified index. `insertPages` is the array of pages with page and `params` properties.
- `pop(opts)` – Navigate back from the current page to the previous page in the navigation stack.
- `popToRoot(opts)` – Navigate back to the root of the navigation stack.
- `remove(startIndex, removeCount, opts)` – Remove the number of `removeCount` pages in the navigation stack starting from a specified index.
- `removeView(viewController, opts)` – Remove specified page from the navigation stack.
- `setRoot(page, params, opts)` – Set the root of the navigation stack.
- `setPages(pages, opts)` – Set all the pages in the navigation stack.

All these methods return a promise that is resolved when the view transition is completed.

Model

Now we are going to implement the next user story to display comments of each top story. For each comment, we also want to show its replies. We'll create a new comments page. Clicking the comments icon of an item in the top stories page triggers the navigation to the comments page. In the comments page, clicking the replies icon navigates to the comments page again, but shows the replies to this comment.

Stories and comments are all items in Hacker News API. They can both have children items, which can be comments or replies. The data structure is recursive, so we can use the same comments page to show both comments and replies.

For the model Item, we add three new properties related to comments.

- `text` – The text of the comment.
- `descendants` – The number of descendants of this item. For stories, this is the number of comments; for comments, this is the number of replies.
- `kids` – The array of ids of descendants of this item. For stories, this is the array of comments ids; for comments, this is the array of replies ids.

Refactoring

When we take a closer look at the comments page, we can see the similarity between the comments page and the top stories page. From the Hacker News API point of view, stories, comments, and replies to comments are all items. These items can be retrieved from the same endpoint `v0/item/<ITEM_ID>`. From the user experiences point of view, the comments and replies page should have all the features as the top stories page, including infinite scrolling and pull-to-refresh. These means most of the code we are using in both pages will be the same. To avoid potential code duplication in the implementation of the comments page, we should do the refactoring of `ItemService` and `TopStoriesPage` first.

Services

We extract the common logic from `ItemService` to an abstract class `AbstractItemService` that will become the parent class of both `ItemService` and the new `CommentService`, see Listing 6-1. Most code in `ItemService` has been pulled into `AbstractItemService`. A new abstract method `getItemIds(itemId: number): Observable<number[]>` is added to get the items ids to load. For the original `ItemService`, this will be the ids of top stories. For the new `CommentService`, this will be the ids of the comments or replies of the specified item. The method `get()` of `AbstractItemService` uses the `getItemIds()` method to get the item ids to work with. The parameter `itemId` is optional and only used by `CommentService`. All the rest of code in method `get()` is copied from the original `ItemService`.

Listing 6-1. AbstractItemService

```
export interface Query {
  refresh?: boolean;
  offset: number;
  limit: number;
}

@Injectable()
export abstract class AbstractItemService {
  queries: Subject<Query> = new Subject<Query>();

  constructor(protected af: AngularFire) {}

  load(query: Query): void {
    // omitting the same code as in ItemService
  }

  get(itemId: number = null): Observable<Items> {
    const rawItemIds = this.getItemIds(itemId);
    // omitting the same code as in ItemService
  }

  abstract getItemIds(itemId: number): Observable<number[]>;
}
```

After the introduction of `AbstractItemService`, we can refactor `ItemService` as in Listing 6-2. The method `getItemIds()` returns the `Observable<number[]>` with code taken from the method `get()` of the original `ItemService`.

Listing 6-2. Refactored ItemService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import { AngularFire } from 'angularfire2';
import { AbstractItemService } from './AbstractItemService';

@Injectable()
export class ItemService extends AbstractItemService {
  constructor(protected af: AngularFire) {
    super(af);
  }

  getItemIds(): Observable<number[]> {
    return this.af.database.list('/v0/topstories')
      .map(ids => ids.map(v => v.$value));
  }
}
```

Now we can easily implement the new `CommentService`. As in Listing 6-3, we only need to implement the method `getItemIds()` that takes an item id as the parameter and returns the `Observable<number[]>` object from the `kids` property of the item object.

Listing 6-3. CommentService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/pluck';
import { AngularFire } from 'angularfire2';
import { AbstractItemService } from './AbstractItemService';

@Injectable()
export class CommentService extends AbstractItemService {
  constructor(protected af: AngularFire) {
    super(af);
  }

  getItemIds(itemId: number): Observable<number[]> {
    return this.af.database.object(`/v0/item/${itemId}`)
      .pluck('kids') as Observable<number[]>;
  }
}
```

Pages

After refactoring the services, we can now do the same refactoring to pages. We pull logic out of `TopStoriesPage` and put into a new abstract class `AbstractItemsPage` with minor changes; see Listing 6-4.

- The constructor only takes arguments of `NavController`, `LoadingController`, and `ToastController` that are required for the common logic. Dependencies to the actual services have been removed.
- Two abstract methods `getItems(): Observable<Items>` and `query(query: Query): void` are added. The method `getItems()` is used to get the items to display. The method `query()` is used to send queries.

Listing 6-4. AbstractItemsPage

```
import { OnInit, OnDestroy } from '@angular/core';
import { NavController } from 'ionic-angular';
import { LoadingController, Loading } from 'ionic-angular';
import { ToastController } from 'ionic-angular';
```

```
import * as concat from 'lodash.concat';
import { Items } from '../model/Items';
import { Query } from '../services/AbstractItemService';
import { Observable, Subscription } from "rxjs";

export abstract class AbstractItemsPage implements OnInit, OnDestroy {
  // omitting the same code as in TopStoriesPage
  constructor(public navCtrl: NavController,
               public loadingCtrl: LoadingController,
               public toastCtrl: ToastController) {
  }

  // omitting the same code as in TopStoriesPage

  abstract getItems(): Observable<Items>;
  abstract query(query: Query): void;
}
```

Listing 6-5 shows the updated code of `TopStoriesPage`. The constructor of `TopStoriesPage` invokes a parent constructor with injected instances of `NavController`, `LoadingController` and `ToastController`. The implementation of methods `getItems()` and `query()` just delegates the requests to the injected `ItemService` object.

Listing 6-5. Updated TopStoriesPage

```
export class TopStoriesPage extends AbstractItemsPage {
  // omitting the same code as in TopStoriesPage
  constructor(public navCtrl: NavController,
               private itemService: ItemService,
               private openPageService: OpenPageService,
               public loadingCtrl: LoadingController,
               public toastCtrl: ToastController) {
    super(navCtrl, loadingCtrl, toastCtrl);
    // omitting the same code as in TopStoriesPage
  }

  ngOnInit(): void {
    super.ngOnInit();
  }

  ngOnDestroy(): void {
    super.ngOnDestroy();
  }

  getItems(): Observable<Items> {
    return this.itemService.get();
  }
}
```

```

    query(query: Query): void {
        this.itemService.load(query);
    }
}

```

View Comments

Even though stories and comments are both items, the components to display them are different. The comment component needs to show the text instead of titles and links. We have separate components to render comments and list of comments.

We use the Ionic CLI to generate stub code of the two components.

```

$ ionic g component comment
$ ionic g component comments

```

CommentComponent

Listing 6-6 shows the template of the CommentComponent. We use binding of the property `innerHTML` to display the content of `item.text` because `item.text` contains HTML markups. We check the property `item.kids` for replies to this comment. If a comment has replies, the number of replies is displayed. Clicking the `` elements of replies count triggers the method `viewReply()` to show the replies.

Listing 6-6. Template of CommentComponent

```

<div *ngIf="!item">
    Loading...
</div>
<div *ngIf="item">
    <div [innerHTML]="item.text"></div>
    <div>
        <span>
            <ion-icon name="person"></ion-icon>
            {{ item.by }}
        </span>
        <span>
            <ion-icon name="time"></ion-icon>
            {{ item.time | timeAgo }}
        </span>
        <span *ngIf="item.kids" (click)="viewReply(item.id)">
            <ion-icon name="chatboxes"></ion-icon>
            {{ item.kids.length }}
        </span>
    </div>
</div>
</div>

```

The `CommentComponent` in Listing 6-7 only has two Input bindings. The property `viewReply` is a function of type `ViewComment` in Listing 6-8.

Listing 6-7. CommentComponent

```
import { Component, Input } from '@angular/core';
import { Item, ViewComment } from '../model/Item';

@Component({
  selector: 'comment',
  templateUrl: 'comment.html'
})
export class CommentComponent {
  @Input() item: Item;
  @Input() viewReply: ViewComment;
}
```

The reason to have a separate type `ViewComment` is the same as we did for the `OpenPage` function type.

Listing 6-8. Function type ViewComment

```
export interface ViewComment {
  (itemId: number): void;
}
```

CommentsComponent

The template of `CommentsComponent` is also like the `ItemsComponent`; see Listing 6-9.

Listing 6-9. Template of CommentsComponent

```
<ion-list *ngIf="items.results.length > 0">
  <ion-item text-wrap *ngFor="let item of items.results">
    <comment [item]="item | async" [viewReply]="viewReply"></comment>
  </ion-item>
</ion-list>
<p *ngIf="items.results.length === 0">
  No items.
</p>
```

The code of `CommentsComponent` is also like the `ItemsComponent`, see Listing 6-10.

Listing 6-10. CommentsComponent

```
import { Component, Input } from '@angular/core';
import { Items } from '../model/Items';
import { ViewComment } from "../model/Item";

@Component({
  selector: 'comments',
  templateUrl: 'comments.html'
})
export class CommentsComponent {
  @Input() items: Items;
  @Input() viewReply: ViewComment;
}
```

Items

We also need to update the ItemComponent to display the number of comments and allows users to view the comments; see Listing 6-11. The method `viewComment()` has the same type `ViewComment` as the method `viewReply` in the `CommentComponent`.

Listing 6-11. Updated template of ItemComponent

```
<span *ngIf="item.descendants" (click)="viewComment(item.id)">
  <ion-icon name="chatboxes"></ion-icon>
  {{ item.descendants }}
</span>
```

View Comments

When the user clicks the comments number in the stories list, we should navigate to the comments page. In the method `viewComment()` of Listing 6-12, we use the method `push()` of `NavController` to perform the navigation. The target of the navigation is `CommentsPage` and the parameter `itemId` to pass is the id of the current item.

Listing 6-12. viewComment

```
viewComment(itemId: number): void {
  this.navCtrl.push(CommentsPage, {
    itemId,
  });
}
```

CommentsPage

The `CommentsPage` in Listing 6-13 also extends the `AbstractItemsPage`. It's injected with an instance of `NavParams` that can be used to get the navigation parameters. In the method `getItems()`, we use `this.params.get('itemId')` to get the value of the parameter `itemId`. Both methods `getItems()` and `query()` delegate to the `CommentService`.

Listing 6-13. CommentsPage

```
import { Component } from '@angular/core';
import { NavController, NavParams } from 'ionic-angular';
import { CommentService } from '../services/CommentService';
import { AbstractItemsPage } from '../AbstractItemsPage';
import { LoadingController, ToastController } from 'ionic-angular';
import { Items } from '../model/Items';
import { Observable } from "rxjs";
import { Query } from '../services/AbstractItemService';

@Component({
  selector: 'page-comments',
  templateUrl: 'comments.html'
})
export class CommentsPage extends AbstractItemsPage {

  constructor(public navCtrl: NavController,
    private params: NavParams,
    private commentService: CommentService,
    public loadingCtrl: LoadingController,
    public toastCtrl: ToastController) {
    super.navCtrl, loadingCtrl, toastCtrl);
    this.viewReply = this.viewReply.bind(this);
  }

  ngOnInit(): void {
    super.ngOnInit();
  }

  ngOnDestroy(): void {
    super.ngOnDestroy();
  }

  viewReply(itemId: number): void {
    this.navCtrl.push(CommentsPage, {
      itemId,
    });
  }
}
```

```

getItems(): Observable<Items> {
    return this.commentService.get(this.params.get('itemId'));
}

query(query: Query): void {
    this.commentService.load(query);
}
}

```

The method `viewReply()` shows the replies to the current comment. It navigates to the same `CommentsPage` with `itemId` set to the id of each reply.

Figure 6-1 shows the screenshot of the comments page.

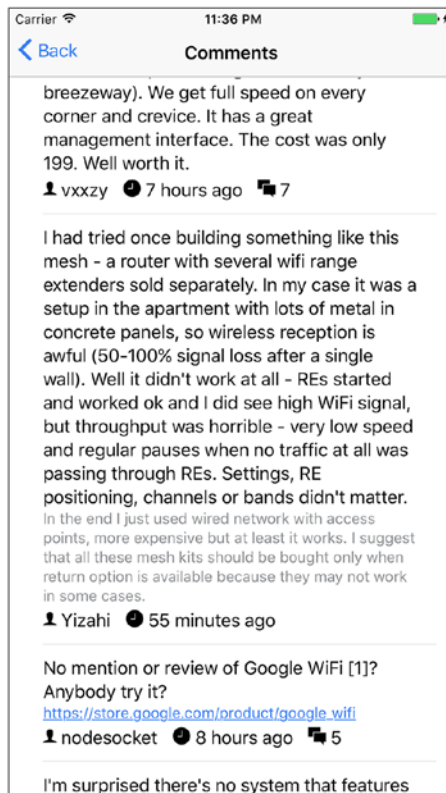


Figure 6-1. Comments page

Summary

In this chapter, we implement the user story to view comments of stories. We use page navigation in Ionic 2 to implement the transition from top stories page to comments page. Most of the code logic in the comments page can be shared with the top stories page, so we refactor components and services from the last chapter to simplify the implementation of the comments page. In the next chapter, we'll discuss user management with Firebase.

User Management

So far, all the implemented user stories in the app are public and available to all users, so now we are going to implement some personalized user stories of the app. Before adding personalization features, we need to add basic support for user management. Implementing user management is a task we must do, but it's tedious and has no additional business value to the app. Although there are a lot of high-quality open source libraries in different programming languages to handle user authentication, we still need to integrate those libraries with app code and manage back-end storage. It's also very common for an app to allow users to log in using third-party providers, for example, Google or Facebook. To support this scenario, more libraries need to be integrated, which makes user management a harder task to finish.

Firebase provides an easy integration with user authentication for the web and mobile apps. Firebase manages user authentication data, so we don't need to manage the storage ourselves. Firebase also supports logging with third-party providers, including Google, Facebook, Twitter, and GitHub.

To support user management, we need to add UI for users to sign up and log in. This requires us to use form controls and other UI components. We start from these Ionic UI controls.

Ionic UI Controls

To gather users' information, we need to use different input controls, including standard HTML form elements like inputs, check boxes, radio buttons, and selects; and components designed for mobile platforms, like toggles or ranges. Ionic provides out-of-box components with beautiful styles for different requirements.

Inputs

The component `ion-input` is for different types of inputs. This component supports the following properties.

- `type` – The type of the input. Possible values are `text`, `password`, `email`, `number`, `search`, `tel`, `url`, `date`, `month`, `time` and `week`.
- `value` – The value of the input.
- `placeholder` – The placeholder of the input.
- `disabled` – Whether the input is disabled or not.
- `clearInput` – Whether to show the icon that can be used to clear the text.
- `clearOnEdit` – Whether to clear the input when the user starts editing the text.

`ion-input` also supports the following events.

- `blur` – Fired when the input loses focus.
- `focus` – Fired when the input has focus.

Below is a basic sample of using `ion-input`.

```
<ion-input type="text" [(ngModel)]="name" name="name" required></ion-input>
```

Check Box

The component `ion-checkbox` creates check boxes with Ionic styles. It has the following properties.

- `color` – The color of the check box. Only predefined color names like `primary` and `secondary` are used.
- `checked` – Whether the check box is checked. The default value is `false`.
- `disabled` – Whether the check box is disabled. The default value is `false`.

`ion-checkbox` also supports the event `ionChange` that fired when the value of the check box is changed.

Below is a basic sample of using `ion-checkbox`.

```
<ion-checkbox [(ngModel)]="enabled"></ion-checkbox>
```

Radio Buttons

Radio buttons can be checked or unchecked. Radio buttons are usually grouped together to allow the user to make selections. A radio button is created using the component `ion-radio`. `ion-radio` supports properties `color`, `checked` and `disabled` with the same meaning as the `ion-checkbox`. `ion-radio` also has property `value` to set the value of the radio button. `ion-radio` supports the event `ionSelect` that's fired when it's selected.

A radio buttons group is created by adding the property `radio-group` to a component, and then all the descendant `ion-radio` components are put into the same group. Only one radio button in the group can be checked. In Listing 7-1, we create a group with three radio buttons. Figure 7-1 shows the screenshot of radio buttons.

Listing 7-1. Radio buttons groups

```
<ion-list radio-group [(ngModel)]="color">
  <ion-list-header>
    Traffic colors
  </ion-list-header>

  <ion-item>
    <ion-label>Red</ion-label>
    <ion-radio value="red"></ion-radio>
  </ion-item>

  <ion-item>
    <ion-label>Green</ion-label>
    <ion-radio value="green"></ion-radio>
  </ion-item>

  <ion-item>
    <ion-label>Blue</ion-label>
    <ion-radio value="blue"></ion-radio>
  </ion-item>
</ion-list>
```

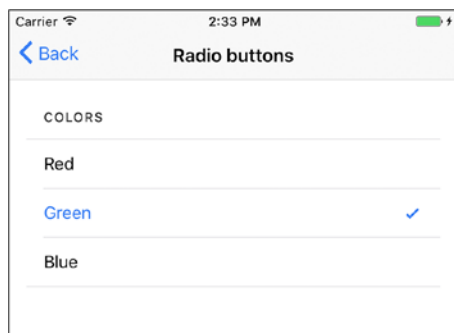


Figure 7-1. Radio buttons

Selects

The component `ion-select` is similar to the standard HTML `<select>` element, but its UI is more mobile-friendly. The options of `ion-select` are specified using `ion-option`. If the `ion-select` only allows single selection, each `ion-option` is rendered as a radio button in the group. If the `ion-select` allows multiple selections, then each `ion-option` is rendered as a check box. Options can be presented using alerts or action sheets. Below are configuration options for `ion-select`.

- `multiple` – Whether the `ion-select` supports multiple selections.
- `disabled` – Whether the `ion-select` is disabled.
- `interface` – Use alerts or action sheets to display the `ion-select`. Possible values are `alert` and `action-sheet`. The default value is `alert`.
- `okText` – The text to display for the OK button.
- `cancelText` – The text to display for the cancel button.
- `placeholder` – The text to display when no selection.
- `selectedText` – The text to display when selected.

`ion-select` also supports the following events.

- `ionChange` – Fired when the selection has changed.
- `ionCancel` – Fired when the selection was canceled.

The `ion-select` in Listing 7-2 renders a single selection select. Figure 7-2 shows the screenshot.

Listing 7-2. *Single selection select*

```
<ion-select [(ngModel)]="color" placeholder="Color">
  <ion-option value="red" selected="true">Red</ion-option>
  <ion-option value="green">Green</ion-option>
  <ion-option value="blue">Blue</ion-option>
</ion-select>
```

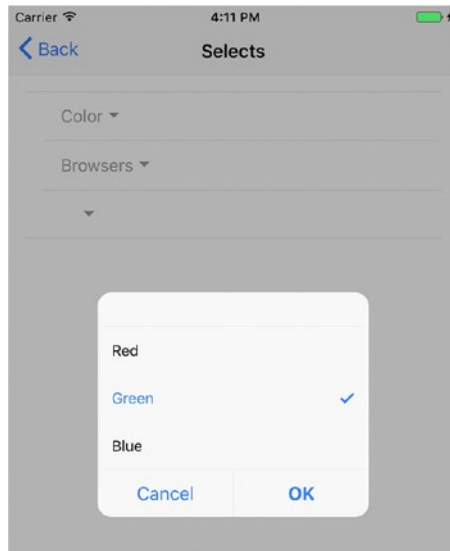


Figure 7-2. *Single selection*

The `ion-select` in Listing 7-3 allows multiple selections. The `ion-options` don't have a property value to set the value, so the value will be the text value. Figure 7-3 shows the screenshot.

Listing 7-3. *Multiple selections select*

```
<ion-select [(ngModel)]="browsers" multiple="true" placeholder="Browsers">
  <ion-option>Chrome</ion-option>
  <ion-option>IE</ion-option>
  <ion-option>Firefox</ion-option>
</ion-select>
```

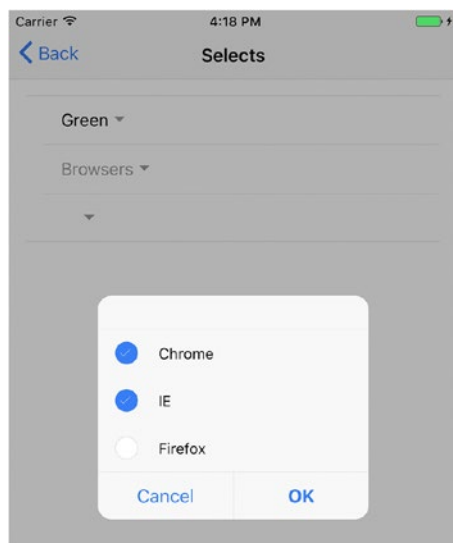


Figure 7-3. Multiple selection

The `ion-select` in Listing 7-4 uses an action sheet to display options.

Listing 7-4. Use action sheet to display

```
<ion-select [(ngModel)]="action" interface="action-sheet">
  <ion-option>Yes</ion-option>
  <ion-option>No</ion-option>
  <ion-option>Maybe</ion-option>
</ion-select>
```

Toggles

Like check boxes, toggles represent Boolean values, but are more user friendly on the mobile platforms. `ion-toggle` supports the same properties and events as `ion-checkbox`. See the code below for a sample of `ion-toggle`.

```
<ion-toggle [(ngModel)]="enabled"></ion-toggle>
```

Ranges

Range sliders allow users to select from a range of values by moving the knobs. By default, a range slider has one knob to select only one value. It also supports using dual knobs to select a lower and upper value. Dual knobs range sliders are perfect controls for choosing ranges, that is, price ranges for filtering.

The component `ion-range` has the following properties. Standard properties, including `color` and `disabled`, are omitted.

- `min` and `max` – Set the minimum and maximum integer value of the range. The default values are 0 and 100, respectively.
- `step` – The value granularity of the range that specifies the increasing or decreasing values when the knob is moved. The default value is 1.
- `snaps` – Whether the knob snaps to the nearest tick mark that is evenly spaced based on the value of `step`. The default value is `false`.
- `pin` – Whether to show a pin with current value when the knob is pressed. The default value is `false`.
- `debounce` – How many milliseconds to wait before triggering the `ionChange` event after a change in the range value. The default value is 0.
- `dualKnobs` – Whether to show two knobs. The default value is `false`.

To add labels to either side of the slider, we can add the property `range-left` or `range-right` to the child components of the `ion-range`. Labels can be texts, icons, or any other components.

The `ion-range` in Listing 7-5 uses two icons as the labels.

Listing 7-5. Labels of `ion-range`

```
<ion-range min="1" max="5" [(ngModel)]="rating">
  <ion-icon range-left name="sad"></ion-icon>
  <ion-icon range-right name="happy"></ion-icon>
</ion-range>
```

The `ion-range` in Listing 7-6 sets the property `step` and `snaps`.

Listing 7-6. Step and snaps

```
<ion-range step="10" snaps="true" pin="true" [(ngModel)]="score">
  <ion-label range-left>Min</ion-label>
  <ion-label range-right>Max</ion-label>
</ion-range>
```


The last `ion-range` in Listing 7-7 has double knobs.

Listing 7-7. Double knobs

```
<ion-range dualKnobs="true" min="0" max="10000" [(ngModel)]= "price">
  <ion-label range-left>Low</ion-label>
  <ion-label range-right>High</ion-label>
</ion-range>
```

Figure 7-4 shows the screenshot for ranges in Listings 7-5, 7-6, and 7-7.

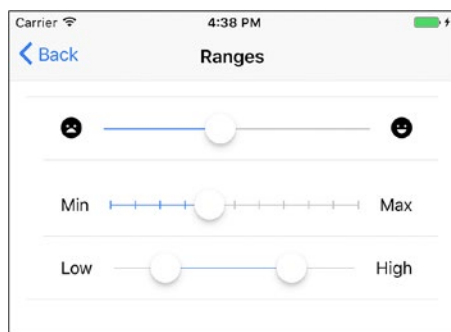


Figure 7-4. Ranges

Labels

Labels can be used to describe different types of inputs. `ion-label` is the component for labels. It supports different ways to position the labels relative to the inputs.

- **fixed** – Labels are always displayed next to the inputs.
- **floating** – Labels will float above the inputs if inputs are not empty or have a focus.
- **stacked** – Labels will always appear on the top of inputs.

We can add the property `fixed`, `floating`, or `stacked` to the `ion-label` to specify the position.

Modal

Modals take up the whole screen space to present the UI to users. Modals are commonly used to gather information from users before they can continue. Modals are created using the method `create(component, data, opts)` of `ModalController`. For the parameters of `create()`, `component` is the

actual component to be presented in the modal, data is a JavaScript object passed to the component, and opts is an options object to configure the modal. The following options are supported when creating modals.

- `showBackdrop` – Whether to show the backdrop or not. The default value is `true`.
- `enableBackdropDismiss` – Whether the modal should be dismissed by tapping on the backdrop. The default value is `true`.
- `enterAnimation` – The animation to use when the modal is presented.
- `leaveAnimation` – The animation to use when the modal is dismissed.

The default values of `enterAnimation` and `leaveAnimation` depend on the target platform.

Toolbar

A toolbar is a generic container for text and buttons. It can be used as a header, subheader, footer, or subfooter. In the previous examples, we use `ion-nav` as the child of `ion-header` to enable page navigation. For modals, we don't need the navigation support, so we can replace the `ion-nav` using `ion-toolbar`.

Buttons in a toolbar should be placed inside of the component `ion-buttons`. We can use different properties to configure the position of the `ion-buttons` inside of the toolbar.

- `start` – On iOS, positioned to the left of the content; On Android and Windows Phone, positioned to the right.
- `end` – On iOS, positioned to the right of the content; on Android and Windows Phone, positioned to the far right.
- `left` – Positioned to the left of all other components.
- `right` – Positioned to the right of all other components.

In Listing 7-8, we have two `ion-buttons` at both sides of the `ion-title`.

Listing 7-8. Toolbar

```
<ion-header>
  <ion-toolbar>
    <ion-buttons left>
      <button ion-button icon-only>
        <ion-icon name="menu"></ion-icon>
      </button>
    </ion-buttons>
    <ion-title>My App</ion-title>
    <ion-buttons right>
      <button ion-button icon-only>
        <ion-icon name="settings"></ion-icon>
      </button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

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

Menu

The app is based on the `sidemenu` template but we didn't use the menu yet. Now we are going to add more information to the menu, including the button to show the login form and details of the currently logged-in user.

The component `ion-menu` should be added as a sibling of the app's content component and linked to the content component by setting the property `content`. The value of `content` should be the name of the local variable that references to the content component. `ion-menu` has the following properties.

- `content` – The reference to the content component.
- `id` – The id of the menu.
- `side` – The side of the view to place the menu. Possible values are `left` and `right`. The default value is `left`.
- `type` – The display type of the menu. Possible values are `overlay`, `reveal` and `push`. The default value depends on the platform.
- `enabled` – Whether the menu is enabled. The default value is `true`.
- `swipeEnabled` – Whether swiping should be enabled to open the menu. The default value is `true`.
- `persistent` – Whether the menu should persist on child pages. The default value is `false`.

menuToggle

The directive `menuToggle` can be added to any button to make it open and close a menu. When the `menuToggle` is placed in a navbar or toolbar, it should be placed as the child of `ion-navbar` or `ion-toolbar`. If `menuToggle` is added to the navbar, the button only appears on the root page, unless the property `persistent` of the `ion-menu` is set to `true`. The value of this directive can be the side or the id of the menu controlled by it.

menuClose

The directive `menuClose` can be added to any button to make it close a menu. The value of this directive can be the side or the id of the menu, just like `menuToggle`.

MenuController

Using `menuToggle` and `menuClose` is generally enough to control the visibility of the menu. However, when there are multiple menus at both sides, or the visibility of the menu depends on complex logic, it's better to use `MenuController`.

The instance of `MenuController` can be injected into the components that want to control the menus. It has the following methods.

- `open(menuId)` – Open the menu with specified id or side.
- `close(menuId)` – Close the menu with specified id or side. If no `menuId` is specified, then all open menus will be closed.
- `toggle(menuId)` – Toggle the menu with specified id or side.
- `enable(shouldEnable, menuId)` – Enable or disable the menu with specified id or side. For each side, when there are multiple menus, only one of them can be opened at the same time. Enabling one menu will also disable other menus on the same side.
- `swipeEnable(shouldEnable, menuId)` – Enable or disable the feature to swipe to open the menu.
- `isOpen(menuId)` – Check if a menu is opened.
- `isEnabled(menuId)` – Check if a menu is enabled.
- `get(menuId)` – Get the Menu instance with specified id or side.
- `getOpen()` – Get the opened Menu instance.
- `getMenus()` – Get an array of all Menu instances.

Email and Password Login

The simplest way to add user management is to allow users to log in using email and password. Firebase already provides built-in support for email and password authentication. Firebase manages storage of users' emails and passwords for the app. By using Firebase's JavaScript library, it's very easy to add email and password login. We need to enable the email and password login for the Firebase project first. In the **Authentication** section of the Firebase project, go to the **Sign-in method** tab and click **Email/Password** to enable it.

Model

We first create a model class for the user; see Listing 7-9. The `User` class has four properties: `uid`, `name`, `email`, `password` and `photoURL`. The property `photoURL` is the URL of the user's profile image. The method `createBlank()` creates a new `User` object with all properties set to empty strings, which can be used to create new `User` objects.

Listing 7-9. User

```
export class User {
  constructor(public uid: string,
               public name: string,
               public email: string,
               public password: string,
               public photoURL?: string) {}

  static createBlank(): User {
    return new User('', '', '', '', '');
  }
}
```

AuthService

As usual, we start with the service for authentication. `AngularFire2` already supports authentication with Firebase using the `auth` object in `AngularFile`. In the previous implementation of the `ItemService`, we already use the injected `AngularFire` object to retrieve data from the Hacker News database. `AngularFire2` simplifies the way to work with Firebase. Now we need to use `AngularFire2` to integrate the authentication for our app. However, `AngularFire2` doesn't provide a way to easily switch between different Firebase instances in the same app. The injected `AngularFire` instance can only easily access a single Firebase instance. To work around this limitation, we use the standard way of `AngularFire` instance to access the app's Firebase instance but use the Firebase database reference directly to access the Hacker News database.

In Listing 7-10, the `HackerNewsService` is the new service to access the Hacker News database. We create a new Firebase config object with the only `databaseURL` set to Hacker News database URL. In the constructor of `HackerNewsService`, we use `firebase.initializeApp()` to initialize a new Firebase app with the config and a new name `HackerNews`. A new name is required because the default name has already been taken by the Firebase app created by `AngularFire2`. We then get a reference to the `firebase.database.Database` object using `app.database()`.

The method `topStories()` is used to get a `FirebaseListObservable` object for the path `/v0/topstories`. The method `item(id: number)` is used to get a `FirebaseObjectObservable` object for the item with the given id. The major difference with `AngularFire2` is that we need to create a `Firebase firebase.database.Reference` object and pass the `Reference` object to `AngularFire2` methods `list()` and `object()`.

Listing 7-10. HackerNewsService

```
import { Injectable } from "@angular/core";
import * as firebase from 'firebase';
import { AngularFire, FirebaseListObservable, FirebaseObjectObservable }
from "angularfire2";

const HACKER_NEWS_API_URL = 'https://hacker-news.firebaseio.com';
const config = {
  databaseURL: HACKER_NEWS_API_URL,
};

@Injectable()
export class HackerNewsService {
  database: firebase.database.Database;
  constructor(private af: AngularFire) {
    const app = firebase.initializeApp(config, 'HackerNews');
    this.database = app.database();
  }

  topStories(): FirebaseListObservable<any> {
    const ref = this.database.ref('/v0/topstories');
    return this.af.database.list(ref);
  }

  item(id: number): FirebaseObjectObservable<any> {
    const ref = this.database.ref(`/v0/item/${id}`);
    return this.af.database.object(ref);
  }
}
```

After we added the `HackerNewsService`, we can update the `ItemService` and `CommentService` to use this new service. Listing 7-11 shows the updated `ItemService`.

Listing 7-11. Updated ItemService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import { AbstractItemService } from '../AbstractItemService';
import { HackerNewsService } from '../services/HackerNewsService';

@Injectable()
export class ItemService extends AbstractItemService {
  constructor(protected hackerNewsService: HackerNewsService) {
    super(hackerNewsService);
  }

  getItemIds(): Observable<number[]> {
    return this.hackerNewsService.topStories()
      .map(ids => ids.map(v => v.$value));
  }
}
```

Listing 7-12 shows the updated `CommentService`.

Listing 7-12. Updated CommentService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/pluck';
import { AbstractItemService } from '../AbstractItemService';
import { HackerNewsService } from '../services/HackerNewsService';

@Injectable()
export class CommentService extends AbstractItemService {
  constructor(protected hackerNewsService: HackerNewsService) {
    super(hackerNewsService);
  }

  getItemIds(itemId: number): Observable<number[]> {
    return this.hackerNewsService.item(itemId)
      .pluck('kids') as Observable<number[]>;
  }
}
```

After we added the new `HackerNewsService`, we can update the `firebaseConfig` in the file `app.module.ts` to use the configurations for the app; see Listing 7-13.

Listing 7-13. Updated Firebase config

```
export const firebaseConfig = {
  apiKey: '<API_KEY>',
  authDomain: '<PROJECT_ID>.firebaseapp.com',
  databaseURL: 'https://<PROJECT_ID>.firebaseio.com',
  storageBucket: '<PROJECT_ID>.appspot.com',
};
```

After the AngularFire2 configuration is updated, we can now use the AngularFire object to implement the AuthService. In Listing 7-14, AuthService has the property user of type BehaviorSubject<User> to represent the current user. BehaviorSubject is a special type of Subject that keeps the last emitted value. Other services can subscribe to the BehaviorSubject to monitor state changes of user authentication. The initial value of the user is null, which means the no user is logged in. this.af.auth is a Observable<FirebaseAuthState> object that emits new values when the user's authentication state is changed. We subscribe to this Observable and convert the FirebaseAuthState values into User objects using the method fromAuthState(). In the method fromAuthState(), we check the property provider to see if it's the AuthProviders.Password, then we get the information of the authenticated user from the property auth. We use this information to create a User object. Because we don't ask the user to upload the profile image, we use the node-gravatar library (<https://github.com/emerleite/node-gravatar>) to generate the Gravatar (<http://en.gravatar.com/>) image based on the email address.

In the method create(), we use this.af.auth.createUser() to create a new user. Two properties, email and password are required to call the method createUser(). These two properties are already included in the User object. The return value of createUser() is a Promise object. In the method login(), we use af.auth.login() to log in the current user. The second argument of login() is an options object to configure the login provider and method. All supported login-in providers and methods are declared in enums AuthProviders and AuthMethods, respectively. In the method logout(), we use af.auth.logout() to log out the current user.

Listing 7-14. AuthService

```
import { Injectable } from "@angular/core";
import { AngularFire, AuthProviders, AuthMethods, FirebaseAuthState }
  from 'angularfire2';
import { User } from '../model/User';
import { BehaviorSubject } from 'rxjs';
import 'rxjs/add/operator/startWith';
import * as find from 'lodash.find';
import * as gravatar from 'gravatar';
import * as firebase from 'firebase';
```



```
@Injectable()
export class AuthService {
  user: BehaviorSubject<User>;
  constructor(private af: AngularFire) {
    this.user = new BehaviorSubject<User>(null);
    this.af.auth
      .subscribe(authState => this.user.next(this.
fromAuthState(authState)));
  }

  create(user: User) {
    return this.af.auth.createUser(user);
  }

  login(user: User) {
    return this.af.auth.login(user, {
      provider: AuthProviders.Password,
      method: AuthMethods.Password,
    })
  }

  logout() {
    this.af.auth.logout();
  }

  fromAuthState(authState: FirebaseAuthState): User {
    if (authState) {
      if (authState.provider == AuthProviders.Password) {
        const user = authState.auth;
        return new User(authState.uid, user.displayName || user.email, user.
email, '', gravatar.url(user.email));
      }
    }
    return null;
  }
}
```

Sign-Up Form

Now we can create the sign-up form for users to create new accounts. The sign-up form contains inputs for name, email, and password.

Let's start from the template of the sign-up component in Listing 7-15. In the `ion-header`, we use the component `ion-toolbar` to show the title and a cancel button. In the form, we use two-way bindings `[(ngModel)]` to bind `ion-inputs` to properties of the user model object. `#signupForm="ngForm"` sets a reference to the `ngForm` object with name `signupForm`, so we can access its state to set the enable status of the sign-up button using `!signupForm.form.valid`.

Listing 7-15. Template of the sign-up form

```

<ion-header>
  <ion-toolbar>
    <ion-title>
      Create New Account
    </ion-title>
    <ion-buttons start>
      <button ion-button (click)="dismiss()">Cancel</button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form #signupForm="ngForm">
    <ion-list>
      <ion-item>
        <ion-label floating>Name</ion-label>
        <ion-input type="text" [(ngModel)]="user.name" name="name"
required></ion-input>
      </ion-item>
      <ion-item>
        <ion-label floating>Email</ion-label>
        <ion-input type="email" [(ngModel)]="user.email" name="email"
required></ion-input>
      </ion-item>
      <ion-item>
        <ion-label floating>Password</ion-label>
        <ion-input type="password" [(ngModel)]="user.password"
name="password" required></ion-input>
      </ion-item>
      <ion-item>
        <button color="primary" ion-button large block round
(click)="signUp()" [disabled]="!signupForm.form.valid">Sign up</button>
      </ion-item>
    </ion-list>
  </form>
</ion-content>

```

The component `SignupPage` in Listing 7-16 extends from the parent class `AbstractBasePage`. The user model object is initialized to an empty `User` object. The method `dismiss()` of `ViewController` can be used to dismiss the modal. When calling the method `dismiss()`, we can pass any data to the component that presents this modal. In the `SignupPage`, we pass a Boolean value to specify whether the user finishes sign-up successfully. In the method `signUp()`, we use the user object to invoke `authService.create()` to create the new user account. If the user account creation is successful, then we dismiss the modal. Otherwise, we use a toast notification to show the error message and keep the modal open.

Listing 7-16. SignupPage

```
import { Component } from '@angular/core';
import {
  NavController,
  ViewController,
  Loading,
  LoadingController,
  ToastController,
} from 'ionic-angular';
import { User } from '../../model/User';
import { AuthService } from '../../services/AuthService';
import { AbstractBasePage } from '../AbstractBasePage';

@Component({
  selector: 'page-signup',
  templateUrl: 'signup.html'
})
export class SignupPage extends AbstractBasePage {
  user: User = User.createBlank();
  constructor(public navCtrl: NavController,
    public viewCtrl: ViewController,
    public loadingCtrl: LoadingController,
    public toastCtrl: ToastController,
    private authService: AuthService) {
    super.navCtrl, loadingCtrl, toastCtrl;
  }

  dismiss(signedUp: boolean = false): void {
    this.viewCtrl.dismiss(signedUp);
  }

  signUp(): void {
    this.showLoading();
    this.authService.create(this.user)
      .then(_ => {
        this.hideLoading();
        this.dismiss(true);
      })
      .catch(error => {
        this.hideLoading();
        this.showError(error);
      });
  }

  protected getLoadingMessage(): string {
    return 'Creating your account...';
  }
}
```

```

    protected getErrorMessage(error: any): string {
        return 'Failed to create your account.';
    }
}

```

The class `AbstractBasePage` in Listing 7-17 encapsulates the logic to show and hide loading indicators, and the logic to show error toast messages. The codes related to these features were in the class `AbstractItemsPage`, and now they were refactored into the new class `AbstractBasePage`. The class `AbstractItemsPage` is also refactored to extend from `AbstractBasePage`. Subclasses of `AbstractBasePage` can override methods `getLoadingMessage()` and `getErrorMessage()` to provide custom messages. The class `AbstractBasePage` is now the base class for all future pages.

Listing 7-17. AbstractBasePage

```

import {
    NavController,
    LoadingController,
    Loading,
    ToastController,
} from "ionic-angular";

export class AbstractBasePage {
    loading: Loading;
    constructor(public navCtrl: NavController,
                public loadingCtrl: LoadingController,
                public toastCtrl: ToastController) {}

    protected showLoading() {
        this.hideLoading();
        this.loading = this.loadingCtrl.create({
            content: this.getLoadingMessage(),
        });
        this.loading.present();
    }

    protected getLoadingMessage(): string {
        return 'Loading...';
    }

    protected hideLoading() {
        if (this.loading) {
            this.loading.dismiss();
            this.loading = null;
        }
    }
}

```

```
protected showError(error: any) {
  this.toastCtrl.create({
    message: this.getErrorMessage(error),
    showCloseButton: true,
  }).present();
}

protected getErrorMessage(error: any): string {
  return '';
}
}
```

Login Page

We create a new page for users to log in. This page contains the form for users to input email and password, and a button to log in. It also contains a button to show the modal for sign-up. In the template of the login page in Listing 7-18, we also use a form to prompt the user to input username and password. It also has a button to show the sign-up form.

Listing 7-18. Template of login page

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Log In
    </ion-title>
    <ion-buttons start>
      <button ion-button (click)="dismiss()">Cancel</button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <form #loginForm="ngForm">
    <ion-list>
      <ion-item>
        <ion-label floating>Email</ion-label>
        <ion-input type="email" [(ngModel)]="user.email" name="email"
required></ion-input>
      </ion-item>
      <ion-item>
        <ion-label floating>Password</ion-label>
        <ion-input type="password" [(ngModel)]="user.password"
name="password" required></ion-input>
      </ion-item>
      <ion-item>
        <button ion-button block round large (click)="login()"
```

```

[disabled]="!loginForm.form.valid">Log In</button>
    </ion-item>
    <ion-item>
        <button ion-button block round color="light" (click)="signUp()">Sign
Up</button>
    </ion-item>
</ion-list>
</form>
</ion-content>

```

The LoginPage in Listing 7-19 also extends from `AbstractBasePage`. In the method `login()`, we use the `AuthService` to log in the current user. A toast notification is displayed in case of login failures. In the method `signUp()`, we create a modal dialog to show the `SignupPage`. We use the method `onDidDismiss()` of the `Modal` object to add a callback function that is invoked when the modal is dismissed. The callback function receives the parameters passed to it when the modal is dismissed. If the sign up finishes successfully, that is, `signedUp` is true, then we invoke the method `dismiss()` to close the login modal dialog, because the newly signed-up user is already logged in automatically.

Listing 7-19. LoginPage

```

import { Component } from '@angular/core';
import {
    NavController,
    ModalController,
    LoadingController,
    Loading,
    ToastController,
    ViewController,
} from 'ionic-angular';
import { SignupPage } from '../pages/signup/signup';
import { User } from '../model/User';
import { AuthService } from '../services/AuthService';
import { AbstractBasePage } from '../AbstractBasePage';

@Component({
    selector: 'page-login',
    templateUrl: 'login.html'
})
export class LoginPage extends AbstractBasePage {
    user: User = User.createBlank();
    constructor(public navCtrl: NavController,
                public modalCtrl: ModalController,
                public loadingCtrl: LoadingController,

```

```
        public toastCtrl: ToastController,
        public viewCtrl: ViewController,
        private authService: AuthService) {
    super(navCtrl, loadingCtrl, toastCtrl);
}

login(): void {
    this.showLoading();
    this.authService.login(this.user).then(_ => {
        this.hideLoading();
        this.dismiss();
    }).catch(error => {
        this.hideLoading();
        this.showError(error);
    });
}

signup(): void {
    const modal = this.modalCtrl.create(SignupPage);
    modal.onDidDismiss(signedUp => {
        if (signedUp) {
            this.dismiss();
        }
    });
    modal.present();
}

dismiss(): void {
    this.viewCtrl.dismiss();
}

protected getLoadingMessage(): string {
    return 'Log in...';
}

protected getErrorMessage(error: any): string {
    return 'Failed to log in. Please check your email and password.';
}
}
```

Menu

Now we update `MyApp` in Listing 7-20 to add code for user management. The instance of `AuthService` is injected, so we can use `AuthService` to get the state of the current user. In the method `showLogin()`, we create a modal to wrap the `LoginPage`, then show the modal.

Listing 7-20. Updated MyApp

```

import { Component, ViewChild } from '@angular/core';
import { Nav, Platform, ModalController } from 'ionic-angular';
import { StatusBar, SplashScreen } from 'ionic-native';
import { AuthService } from '../services/AuthService';
import { TopStoriesPage } from '../pages/top-stories/top-stories';
import { LoginPage } from '../pages/login/login';

@Component({
  templateUrl: 'app.html'
})
export class MyApp {
  @ViewChild(Nav) nav: Nav;

  rootPage: any = TopStoriesPage;

  constructor(public platform: Platform,
               public modalCtrl: ModalController,
               public authService: AuthService) {
    this.initializeApp();
  }

  initializeApp() {
    this.platform.ready().then(() => {
      StatusBar.styleDefault();
      SplashScreen.hide();
    });
  }

  showTopStories() {
    this.nav.setRoot(TopStoriesPage);
  }

  showLogIn() {
    const modal = this.modalCtrl.create(LoginPage);
    modal.present();
  }

  logOut() {
    this.authService.logOut();
  }
}

```

Then we update the side menu to reflect current user's status, see Listing 7-21. If no user is logged in, that is, the current emitted value of `authService.user` is null, then the menu item for login is displayed, otherwise, the menu item for logout is displayed. If the user is logged in, we also show the user's emails and photo. The `authService.user` is a `BehaviorSubject` object, so we use the `async` pipe to extract its value.

Listing 7-21. Updated template of side menu

```
<ion-menu [content]="content">
  <ion-header>
    <ion-toolbar>
      <ion-title>Hacker News</ion-title>
    </ion-toolbar>
  </ion-header>

  <ion-content>
    <ion-list>
      <ion-item *ngIf="authService.user | async">
        <ion-avatar *ngIf="(authService.user | async)?.photoURL" item-left>
          <img [src]="(authService.user | async)?.photoURL">
        </ion-avatar>
        <h2>{{ (authService.user | async)?.name }}</h2>
      </ion-item>
      <button id="btnShowTopStories" menuClose ion-item
(click)="showTopStories()">Top Stories</button>
      <button id="btnShowLogin" *ngIf="!(authService.user | async)"
menuClose ion-item (click)="showLogIn()">Log In</button>
      <button id="btnLogout" *ngIf="authService.user | async" menuClose
ion-item (click)="logOut()">Log Out</button>
    </ion-list>
  </ion-content>

</ion-menu>

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

Update User's Name

In the sign-up form, we asked the user to input both name and email, but when the user is logged in using email, the name was not displayed. This is because we didn't save the user's name after the sign-up. Firebase also stores a user's basic profile information, including the user's display name and the profile photo URL. We need to update the user's profile to set the display name after sign-up. However, the AngularFire2 library doesn't expose user profile-related features in its auth object, so we need to interact with Firebase JavaScript SDK directly.

In Listing 7-22, the `FirebaseApp` from AngularFire2 is actually the `firebase.app.App` object that refers to the current Firebase instance. In the `AuthService`, we use `@Inject(FirebaseApp)` to inject the `firebase.app.App` object. In the method `create()`, `this.app.auth()` gets the `firebase.auth.Auth` object. The method `createUserWithEmailAndPassword()` creates a new

user and returns a Promise with the new `firebase.User` object, then the method `updateProfile()` of the `firebase.User` object is invoked to update the user's display name.

Listing 7-22. Update user name

```
@Injectable()
export class AuthService {
  constructor(private af: AngularFire,
               @Inject(FirebaseApp) private app: any) {

  }

  create(user: User) {
    return this.app.auth().createUserWithEmailAndPassword(user.email, user.
      password)
      .then(u => u.updateProfile({
        displayName: user.name,
      }));
  }
}
```

Third-Party Login

It's tedious for users to register different accounts for various online services or apps. So many services or apps allow users to log in using existing accounts of popular third-party services, for example, Twitter or Facebook. Apart from the email/password authentication, Firebase provides support for common third-party providers, including Google, Twitter, Facebook, and GitHub. It's very easy to add support for logging in using third-party providers.

Before we can use those third-party service providers, we need to enable them in Firebase console. In the **Sign-in method** tab of the **Authentication** section, we can enable all supported sign-in providers.

- Google sign-in can be enabled without additional settings.
- For Twitter sign-in, we need to register the app on Twitter and get the API key and secret for the app. When enabling Twitter sign-in, we need to fill the API key and secret. After it's enabled, we set https://hacker-news-ionic2.firebaseio.com/__/auth/handler as the Callback URL for the Twitter app.

- Facebook sign-in is similar with Twitter sign-in; we also need to register the app on Facebook to get the app ID and secret to enable it. After enabling it, we also need to set the OAuth redirect URI to the same as Twitter app's Callback URL.
- GitHub sign-in is also very similar. We first register the app on GitHub to get the client id and secret, and then enable it in the Firebase console, and finally the Authorization callback URL of the GitHub app.

Google Login

For web apps, it's very easy to add these login methods. For example, if we want to enable Google login, in the method `loginWithGoogle()` in Listing 7-23, we use the method `login()` of `this.af.auth` object to log in with Google. The login method is `AuthMethods.Popup`, so a browser window is popped up to Google's login page and asks the user to authorize the access. After the access is granted, the pop-up window is closed and we can get the authentication information from the observable `this.af.auth`.

Listing 7-23. Google login for web apps

```
loginWithGoogle() {  
  return this.af.auth.login({  
    provider: AuthProviders.Google,  
    method: AuthMethods.Popup,  
  });  
}
```

When we do the testing on browsers, this pop-up authentication method works fine. But when we try to run the app on emulators or devices, we find it failed with error code `auth/operation-not-supported-in-this-environment`. To enable support for actual devices, we need to use native Cordova plugins. The plugin for Google login is `cordova-plugin-googleplus` (<https://github.com/EddyVerbruggen/cordova-plugin-googleplus>).

Before we use this Cordova plugin, we need to add apps for the Firebase project. This can be done in the **Overview** page on the Firebase console. The configuration for this plugin is different for iOS and Android platforms.

iOS

First, we add a new iOS app in the Firebase console. In the pop-up dialog, we need to input the iOS bundle ID, which is the id in the `config.xml` file. Then we can download the `GoogleService-Info.plist` from the dialog. This

file needs to be copied into iOS app directory under `platforms/ios`. After that, we use Xcode to open the iOS workspace and add the `GoogleService-Info.plist` to the project.

We need to find the `REVERSED_CLIENT_ID` in the `GoogleService-Info.plist` file that has the pattern like `com.googleusercontent.apps.<APP_ID>`. This `REVERSED_CLIENT_ID` is required when installing the `cordova-plugin-googleplus` plugin using `cordova CLI`.

```
$ cordova plugin add cordova-plugin-googleplus --save --variable REVERSED_CLIENT_ID=<REVERSED_CLIENT_ID>
```

Android

On Android, we need to get the SHA-1 fingerprint of the app's signing certificate first. You can find more details in the official doc (<https://developers.google.com/android/guides/client-auth>). For debug builds, the fingerprint can be extracted from the default Android debug keystore in `~/.android/debug.keystore`. For release builds, the fingerprint can be extracted from the production keystore.

```
$ keytool -exportcert -list -v -alias androiddebugkey -keystore
~/.android/debug.keystore
```

From the output of the command above, we need to copy the SHA-1 fingerprint. In the Firebase console, we add an Android app for the same Firebase project. In the add dialog, we need to provide the package name, a nickname, and the SHA-1 fingerprint. The package name is the id in the `config.xml` that is the same as the bundle ID in the iOS app.

If you want to develop and test on different machines, you need to run the `keytool` command above to extract SHA-1 fingerprints on those machines and add them in the **Settings** page of the Android app in Firebase console. If you got the 12501 error, this is likely because the fingerprint on your local machine was not added correctly.

Login

Now we can use the `cordova-plugin-googleplus` plugin to log in. There are two phases to log in with Google. The first phase is to use the plugin to log into Google, the second phase is to log into Firebase. In the first phase, the user needs to log in with a Google account and grant the permission to allow the app to access its public profile. If the first phase finishes successfully, the response is an object that looks like Listing 7-24.

Listing 7-24. Google login success response

```
{
  "userId": "",
  "displayName": "Fu Cheng",
  "refreshToken": "",
  "email": "alexcheng1982@gmail.com",
  "serverAuthCode": "",
  "accessToken": "",
  "imageUrl": "",
  "idToken": ""
}
```

We need to use the `idToken` in the second phase. The `idToken` is the credential to be used with Firebase. We add a new method `loginWithCredential()` in `AuthService` to support login with credentials; see Listing 7-25. `loginWithCredential()` uses `firebase.auth.AuthCredential` object as the credential and `AuthMethods.OAuthToken` as the login method.

Listing 7-25. loginWithCredential

```
loginWithCredential(credential: firebase.auth.AuthCredential,
                    provider: AuthProviders) {
  return this.af.auth.login(credential, {
    provider,
    method: AuthMethods.OAuthToken,
  });
}
```

In the `LoginPage`, we add the new method `loginWithGoogle()` to start the login process with Google; see Listing 7-26. Ionic Native provides a wrapper API for the plugin `cordova-plugin-googleplus` in the object `GooglePlus`. The method `login()` accepts three arguments. The first argument is the options object. `scopes` is the authentication scopes, “profile email” is the default value. `webClientId` can be found on the **Settings** page of the Google authentication method in the Firebase console. `offline` is set to `false` because we don’t need offline access. The second argument is the success callback function. We already see the format of the input argument response in Listing 7-24. We create the credential object using `firebase.auth.GoogleAuthProvider.credential()` and pass it to `loginWithCredential()` for login. The last argument is the error callback function.

Listing 7-26. Log in with Google

```
loginWithGoogle() {
  this.showLoading();
  GooglePlus.login({
    scopes: 'profile email',
```

```

webClientId: '<APP_ID>.apps.googleusercontent.com',
offline: false,
}).then(response => {
  const credential = firebase.auth.GoogleAuthProvider.
    credential(response.idToken);
  this.authService.signInWithCredential(credential, AuthProviders.Google)
    .then(_ => {
      this.hideLoading();
      this.dismiss();
    })
    .catch(error => {
      this.hideLoading();
      this.showError(error);
    })
  }).catch(message => {
    this.hideLoading();
    this.showError({ message });
  });
}

```

After the third-party login is enabled, we need to update `fromAuthState()` in `AuthService` to handle authentication state for third-party providers. In the `FirebaseAuthState` object, if the value of the property `provider` is one of the four third-party providers, then the user information is stored in the corresponding property. For example, if the value of `provider` is `AuthProviders.Google`, then the property of user information is `google`. The type of this property is `firebase.UserInfo`, which can be used to extract user information. In Listing 7-27, `providers` is the mapping from enum values in `AuthProviders` to the property name in the `FirebaseAuthState` object. Here we use the `photoURL` provided by third-party providers and fall back to Gravatar.

Listing 7-27. Update `fromAuthState()` for third-party providers

```

export const providers = {
  [AuthProviders.Google]: 'google',
  [AuthProviders.Twitter]: 'twitter',
  [AuthProviders.Facebook]: 'facebook',
  [AuthProviders.Github]: 'github',
};

fromAuthState(authState: FirebaseAuthState): User {
  if (authState) {
    if (authState.provider == AuthProviders.Password) {
      const user = authState.auth;
      return new User(authState.uid, user.displayName || user.email,
        user.email, '', gravatar.url(user.email));
    } else if (authState.provider in providers) {

```

```
const user = authState[providers[authState.provider]] as
  firebase.UserInfo;
return new User(authState.uid, user.displayName, user.email, '',
  user.photoURL || gravatar.url(user.email));
}
}
return null;
}
```

Figure 7-5 shows the screenshot of login with Google on a real device.

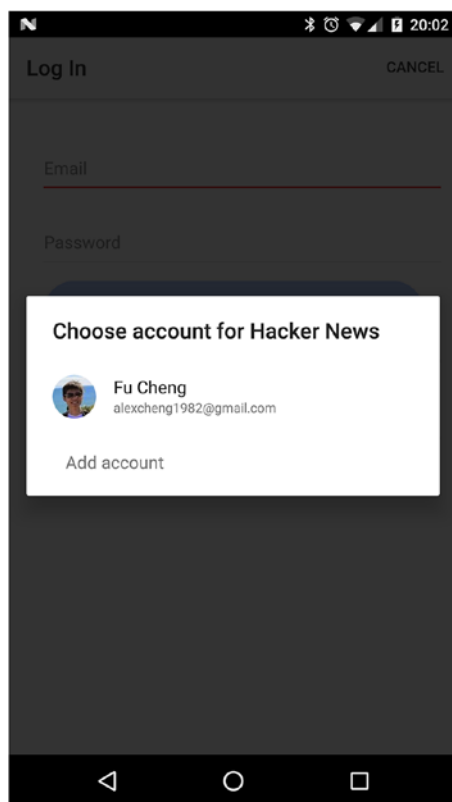


Figure 7-5. Login with Google

Facebook Login

To enable Facebook login for the app, we need to create a new app first in Facebook developers site (<https://developers.facebook.com>). In the **Basic** page of the **Settings** group of the created app, we can see the App

ID and App Secret. Use these two values to enable the Facebook login method in the **Sign-in Method** section of the **Authentication** page in the Firebase console. Go back to the Facebook app, in the **Products** section, add **Facebook Login**. In the **Settings** page of Facebook Login, use the URL https://hacker-news-ionic2.firebaseio.com/__/auth/handler as the **OAuth redirect URIs**. In the **Basic** page of the **Settings** group, use the **Add Platform** button to add iOS and Android platform. The app id in config.xml is used as the Bundle ID for iOS and Google Play Package Name for Android.

Like Google login, we need a Cordova plugin for Facebook login. The plugin we use is cordova-plugin-facebook4 (<https://github.com/jeduan/cordova-plugin-facebook4>). We use Cordova CLI to install the plugin. We need to provide the Facebook App ID during the installation.

```
$ cordova plugin add cordova-plugin-facebook4 --save --variable
APP_ID="<APP_ID>" --variable APP_NAME="<APP_NAME>"
```

After the plugin is installed, we can use its API to log in with Facebook. Ionic Native provides a wrapper for this plugin, we can use it to simplify the code to access the API. Listing 7-28 shows the code to log in with Facebook. The object Facebook is imported from ionic-native. The method login() accepts one parameter, which is an array of permissions (<https://developers.facebook.com/docs/facebook-login/permissions>) to ask from the user. Here we only require the permission public_profile to access user's public profile information, which is also the default permission. The return value of login() is a Promise object. If the login is successful, then from the login response, we can get the access token using response.authResponse.accessToken. A credential object is created from the access token using firebase.auth.FacebookAuthProvider.credential(). Finally, we use signInWithCredential() of the AuthService to do the login.

Listing 7-28. Log in with Facebook

```
loginWithFacebook() {
  this.showLoading();
  Facebook.login(['public_profile']).then(response => {
    const credential = firebase.auth.FacebookAuthProvider.
      credential(response.authResponse.accessToken);
    this.authService.signInWithCredential(credential, AuthProviders.Facebook)
      .then(_ => {
        this.hideLoading();
        this.dismiss();
      }).catch(error => {
        this.hideLoading();
        this.showError(error);
      });
  });
}
```



```

}).catch(message => {
  this.hideLoading();
  this.showError({ message });
});
}

```

Figure 7-6 shows the screenshot of login with Facebook.

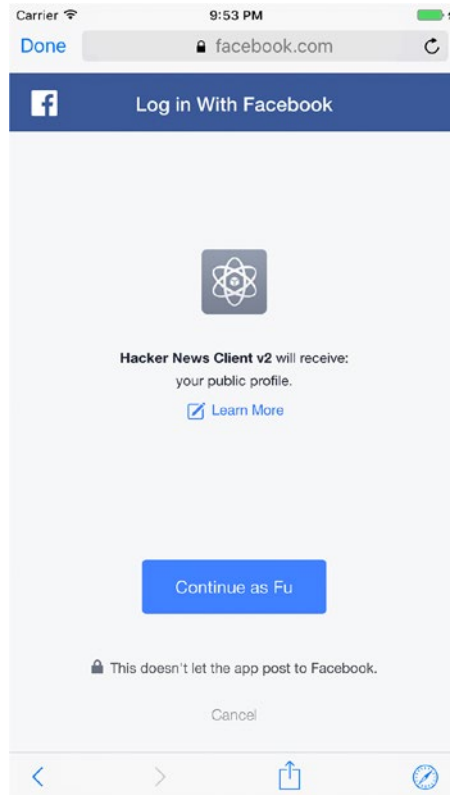


Figure 7-6. Login with Facebook

GitHub Login

For Google and Facebook login, it's convenient to use Cordova plugins to handle the authentication flow and provide the token for login with Firebase. However, for other third-party providers, for example, Twitter or GitHub, there are no Cordova plugins available. Most of these third-party providers use OAuth (<https://oauth.net/>) as the authorization protocol. We can

process the OAuth flow manually and get the OAuth access token for login with Firebase. We use GitHub (<https://developer.github.com/v3/oauth/>) as the example to demonstrate the process.

Similar with Google and Facebook, we need to create a new OAuth app (<https://github.com/settings/applications/new>) in GitHub first. The **Authorization callback URL** of the app should be set to `http://localhost/callback`. Then we need to enable the sign-in method GitHub in Firebase console with the client id and secret of the newly created GitHub app.

The first step of OAuth flow is to open the URL for user to grant access. In Ionic 2 apps, this is done by using the plugin `InAppBrowser`. We use the library `ng2-cordova-oauth` (<https://github.com/nraboy/ng2-cordova-oauth>) to handle this. The package `ng2-cordova-oauth` is installed using `yarn`.

```
$ yarn add ng2-cordova-oauth
```

Then in the file `app.module.ts`, we declare the provider `OAuth` for dependency injection; see Listing 7-29.

Listing 7-29. Configure ng2-cordova-oauth

```
import { OAuthCordova } from 'ng2-cordova-oauth/platform/cordova';
import { OAuth } from 'ng2-cordova-oauth/oauth';

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    ...
  ],
  providers: [
    ...,
    { provide: OAuth, useClass: OAuthCordova }
  ]
})
export class AppModule {}
```

To continue the OAuth process, we need to send a HTTP POST request to GitHub. We use another Cordova plugin `cordova-HTTP` (<https://github.com/wymsee/cordova-HTTP>) for sending HTTP requests. Use the command below to install this plugin.

```
$ cordova plugin add cordova-plugin-http --save
```

In Listing 7-30, the object `GitHubInfo` contains information about the GitHub app. The class `GitHub` is the `OAuthProvider` implementation to be used with the package `ng2-cordova-oauth`. The property `state` is a random string that can be used to check if the request comes from your app. In the method `loginWithGitHub()`, the method `this.oauth.loginVia()` starts the OAuth flow by opening a browser window for the user to approve the request. We can extract the code from the response after the user grants the access. The method `cordovaHTTP.post()` sends a HTTP POST request to GitHub and get the access token. A credential object is created from the access token using the method `firebase.auth.GithubAuthProvider.credential()`. The credential object is then used in the method `authService.loginWithCredential()` for login with Firebase.

Listing 7-30. Login with GitHub

```
import { OAuth } from 'ng2-cordova-oauth/oauth';
import { OAuthProvider } from 'ng2-cordova-oauth/provider';
import get from 'lodash.get';

declare var window:any;

const GitHubInfo = {
  clientId: '<client id>',
  clientSecret: '<client secret>',
  state: '<random string>',
};

class GitHub extends OAuthProvider {
  protected authUrl: string = 'https://github.com/login/oauth/authorize';
  protected defaults: Object = {
    responseType: 'code'
  };
}

export class LoginPage extends AbstractBasePage {
  githubProvider: GitHub = new GitHub({
    clientId: GitHubInfo.clientId,
    appScope: ['user:email'],
    state: GitHubInfo.state,
  });
}
```

```

logInWithGitHub() {
  this.showLoading();
  this.oauth.logInVia(this.githubProvider).then(response => {
    window.cordovaHTTP.post('https://github.com/login/oauth/access_token', {
      client_id: GitHubInfo.clientId,
      client_secret: GitHubInfo.clientSecret,
      code: get(response, 'code'),
      state: GitHubInfo.state,
    }, {
      Accept: 'application/json',
    },
    response => {
      const data = JSON.parse(response.data);
      const credential = firebase.auth.GithubAuthProvider.
        credential(data.access_token);
      this.authService.logInWithCredential(credential, AuthProviders.
        Github)
        .then(_ => {
          this.hideLoading();
          this.dismiss();
        }).catch(error => {
          this.hideLoading();
          this.showError(error);
        });
    },
    response => {
      this.hideLoading();
      this.showError({
        message: response.error,
      });
    }
  });
}, error => {
  this.hideLoading();
  this.showError(error);
});
}
}

```

Figure 7-7 shows the screenshot of login with GitHub.

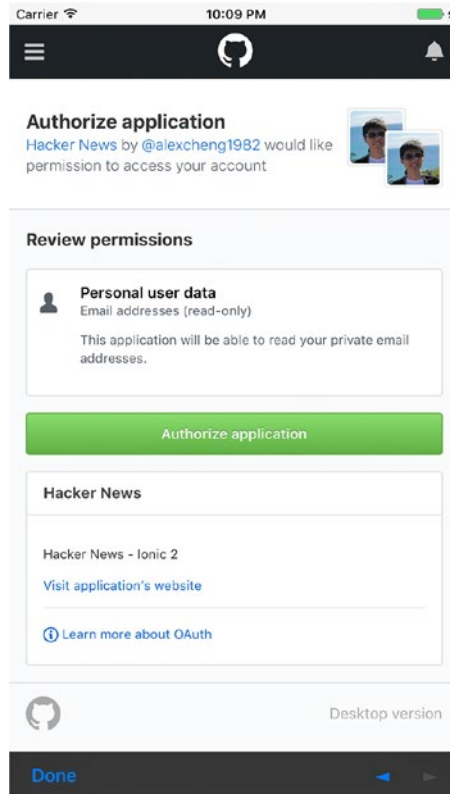


Figure 7-7. Login with GitHub

Test

We need to add test suites for the sign-up form and login page. We can use the Jasmine spy to intercept calls to methods in AuthService. For example, we can use following code to simulate success or failure login requests.

```
spyOn(authService, 'login').and.returnValue(Promise.resolve(true));
spyOn(authService, 'login').and.returnValue(Promise.reject(false));
```

The complete code can be found in the source code with this book.

Summary

In this chapter, we discuss how to add user management to the example app with Firebase. For email/password login, we use Ionic 2 forms for users to sign up and log in. For third-party login, we use Cordova plugins to support login with Google and Facebook. We also demonstrate how to manually process OAuth flow for GitHub. In the next chapter, we'll implement the user story to manage users' favorites.

Manage Favorites

After adding the user management, we can now move to the next user story that allows users to add stories to their favorites.

Favorites Service

Before we start to implement the favorites service, we need to design the storage structure in the Firebase database. The root path for favorites data will be `/favorites`. Each user should have its own favorites, so we use the user's uid as the key for the user's data. The path for each user's favorites data is `/favorites/{uid}`. For each item that the user adds to the favorites, we need to store the item's id and the timestamp. The timestamp is used to sort the favorite items to make sure newly added items appear first in the list. Then we have two options to store the favorites data. The first option is to store favorite items as an array with each element containing the item id and timestamp. The second option is to store favorite items as an object with the item id as the property and the timestamp as the value. To find out which option is better, we need to consider our usage scenarios of favorites data.

- Add an item to favorites or remove an item from favorites. Both options can support this scenario. However, using an array makes the removal of favorite items much harder. When pushing a new element to a Firebase array, Firebase assigns a unique key to it. This key is required when removing the element from the array. We need to pass the key to the UI to remove an item from favorites. When using the second option, we don't need a separate key as the item id is enough.

- Get a list of all favorite items. The first option is better for this scenario. For the second option, because favorite items are stored as an object, we need to convert the object into an array by enumerating the properties.
- Check whether an item is in the favorites. This scenario is very easy to support with the second option. We only need to check the existence of data at the path for that item id. If the first option is used, we need to iterate the whole array to check the existence of specified item id. A more severe issue with the array is that we can only use one `Observable` instance to represent all the favorite items. This `Observable` instance will be merged with the `Observable` instance for items to generate the final `Observable` instance. In this case, any changes to the favorite items `Observable` will trigger the change of all items. This is unnecessary and has serious performance issues.

After considering above scenarios, the final choice is to use the object structure.

In Listing 8-1, like `ItemService`, `FavoriteService` class also extends from `AbstractItemService`, because favorite items are displayed in the same way as top stories. `FavoriteService` also has the same features as `ItemService`. The method `add()` adds an item to the favorites by its id. We get the current user from the `BehaviorSubject<User>` object of `AuthService`. If the user is null, we simply return a resolved `Promise` object. Otherwise, we first use `getItemPath()` to get the path of this favorite item, then use this. `af.database.object()` to create the `FirebaseObjectObservable` object for this path, and finally use `set()` to set the value to the current timestamp. The return result of the `set()` is a `Promise` object.

The method `remove()` removes an item from favorites. It follows the same pattern as `add()`, but use the method `remove()` of the `FirebaseObjectObservable` object to remove the value.

The method `getItemIds()` returns an `Observable` object of an array of favorite item ids. This is done by a `mergeMap` operation on the `BehaviorSubject<User>` object. If the user is null, that is, no user is logged in, an empty array is returned. Otherwise, we get the `FirebaseObjectObservable` object that contains all favorite items. The `map` operation turns the object into an array of elements with `id` and `timestamp` properties, then `filter` operation filters out those non-numerical ids. After that, `sortBy` operation sorts the array by timestamp. Because the result of the `sortBy` operation is in ascending order, we use `reverse()` to make the latest favorite items appear first. The final `pluck` operation extracts only the property `id` from the elements in the array.

The method `getFavoriteItem()` returns an `Observable` of a favorite item. If the user is logged in, we just return the `FirestoreObjectObservable` object for this favorite item. If the item is in the favorites, the property `$value` of the object contains the timestamp, or `$value` is null. We use this method to determine whether an item is in favorites.

Listing 8-1. Favorites service

```
import { Injectable } from '@angular/core';
import { AngularFire } from 'angularfire2';
import { AuthService } from '../AuthService';
import { AbstractItemService } from '../AbstractItemService';
import { HackerNewsService } from '../services/HackerNewsService';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/mergeMap';
import * as sortBy from 'lodash.sortby';
import * as map from 'lodash.map';
import * as pluck from 'lodash.pluck';
import * as filter from 'lodash.filter';

@Injectable()
export class FavoriteService extends AbstractItemService {
  constructor(private af: AngularFire,
               private authService: AuthService,
               protected hackerNewsService: HackerNewsService) {
    super(hackerNewsService);
  }

  add(itemId: number) {
    const user = this.authService.user.getValue();
    if (user == null) {
      return Promise.resolve(null);
    }
    return this.af.database.object(this.getItemPath(user.uid, itemId)).
      set(new Date().getTime());
  }

  remove(itemId: number) {
    const user = this.authService.user.getValue();
    if (user == null) {
      return Promise.resolve(null);
    }
    return this.af.database.object(this.getItemPath(user.uid, itemId)).
      remove();
  }
}
```

```

getItemIds(): Observable<number[]> {
    return this.authService.user.mergeMap(user => {
        if (user == null) {
            return Observable.of([]);
        }
        return this.af.database.object(this.getBasePath(user.uid))
            .map(obj => pluck(sortBy(filter(map(obj, (v, k) => ({
                id: parseInt(k, 10),
                timestamp: v,
            }))), v => !isNaN(v.id)), 'timestamp').reverse(), 'id'));
    });
}

getFavoriteItem(itemId: number): Observable<any> {
    return this.authService.user.mergeMap(user => {
        if (user) {
            return this.af.database.object(this.getItemPath(user.uid, itemId));
        } else {
            return Observable.of({
                $value: null,
            });
        }
    });
}

private getBasePath(uid: string): string {
    return `~/favorites/${uid}`;
}

private getItemPath(uid: string, itemId: number): string {
    return `${this.getBasePath(uid)}/${itemId}`;
}
}

```

Favorites Page

The page to list all favorite items is basically the same as the top stories page. The only difference is that the favorites page only list items from the favorites. We can reuse most of the code in the `TopStoriesPage`. Before we create the favorites page, we need to refactor current `TopStoriesPage` to extract common logic into a parent class, so both pages can inherit from the same class.

The new `AbstractFavoriteItemsPage` class extends from class `AbstractItemsPage`; see Listing 8-2. The property `openPage` and method `viewComment()` comes from original `TopStoriesPage`. The new property `enableFavorite` is an `Observable<boolean>` object that contains the

value of whether the favorites feature should be enabled. The favorites feature requires the user to log in, so `enableFavorite` is a direct mapping from the `authService.user`. The methods `addToFavorite()` and `removeFromFavorite()` simply delegate to corresponding methods in the `FavoriteService`. `AbstractFavoriteItemsPage` class has a new abstract method `getRawItems()` to get the `Observable<Items>` object that contains the items data. The inherited method `getItems()` from `AbstractItemsPage` now uses the method `mergeFavorites()` to combine the favorites data.

For the top stories and favorites list page, we need to combine the items data from Hacker News database and the favorites data from the app's database to create the final item objects for render. The method `mergeFavorites()` is used for the combination. `this.getRawItems()` returns the original `Observable<Items>` object from `HackerNewsService`. The actual items are contained in the results array of `Items` object, so we need to use `map` operator to update the property results. For each `Observable<Item>` element in the array results, we use a `mergeMap` operator to merge with the result of `getFavoriteItem()` of `FavoriteService`. The value of `Observable` from `getFavoriteItem()` is an object with property `$value`. We can set the value of property `isFavorite` based on the value of `$value`. The property `isFavorite` is newly added to model `Item` to represent if an item is in the favorites.

Listing 8-2. AbstractFavoriteItemsPage

```
import { Observable } from "rxjs";
import { AbstractItemsPage } from '../AbstractItemsPage';
import { Items } from '../model/Items';
import { AuthService } from '../services/AuthService';
import { FavoriteService } from '../services/FavoriteService';
import { NavController, LoadingController, ToastController } from
'ionic-angular';
import { CommentsPage } from '../comments/comments';
import { OpenPage } from "../model/Item";
import { OpenPageService } from "../services/OpenPageService";

export abstract class AbstractFavoriteItemsPage extends AbstractItemsPage {
  openPage: OpenPage;
  enableFavorite: Observable<boolean>;
  constructor(public navCtrl: NavController,
               public loadingCtrl: LoadingController,
               public toastCtrl: ToastController,
               protected openPageService: OpenPageService,
               protected authService: AuthService,
               protected favoriteService: FavoriteService) {
    super.navCtrl, loadingCtrl, toastCtrl);
    this.openPage = openPageService.open.bind(openPageService);
    this.viewComment = this.viewComment.bind(this);
    this.addToFavorite = this.addToFavorite.bind(this);
```

```
this.removeFromFavorite = this.removeFromFavorite.bind(this);
this.enableFavorite = authService.user.map(user => user != null);
}

protected viewComment(itemId: number): void {
  this.navCtrl.push(CommentsPage, {
    itemId,
  });
}

protected addToFavorite(itemId: number) {
  return this.favoriteService.add(itemId);
}

protected removeFromFavorite(itemId: number) {
  return this.favoriteService.remove(itemId);
}

private mergeFavorites(): Observable<Items> {
  return this.getRawItems().map(items => {
    items.results = items.results.map(result =>
      result.mergeMap(item => this.favoriteService.getFavoriteItem
        (item.id).map(v => {
          item.isFavorite = !!v.$value;
          return item;
        })))
  });
  return items;
});
}

protected getItems(): Observable<Items> {
  return this.mergeFavorites();
}

protected abstract getRawItems(): Observable<Items>;
}
```

The FavoritesPage can be very simple. It just extends from class AbstractFavoriteItemsPage and implements the method `getRawItems()` to return the items from FavoriteService. See Listing 8-3.

Listing 8-3. FavoritesPage

```
import { Component } from '@angular/core';
import { NavController, LoadingController, ToastController } from
'ionic-angular';
import { Observable } from "rxjs";
import { AuthService } from '../services/AuthService';
```

```

import { FavoriteService } from '../..services/FavoriteService';
import { Items } from '../..model/Items';
import { Query } from '../..services/AbstractItemService';
import { AbstractFavoriteItemsPage } from '../AbstractFavoriteItemsPage';
import { OpenPageService } from "../../services/OpenPageService";

@Component({
  selector: 'page-favorites',
  templateUrl: 'favorites.html'
})
export class FavoritesPage extends AbstractFavoriteItemsPage {
  constructor(public navCtrl: NavController,
               public loadingCtrl: LoadingController,
               public toastCtrl: ToastController,
               protected openPageService: OpenPageService,
               protected authService: AuthService,
               protected favoriteService: FavoriteService) {
    super(navCtrl, loadingCtrl, toastCtrl, openPageService, authService,
          favoriteService);
  }

  ngOnInit(): void {
    super.ngOnInit();
  }

  ngOnDestroy(): void {
    super.ngOnDestroy();
  }

  getRawItems(): Observable<Items> {
    return this.favoriteService.get();
  }

  query(query: Query): void {
    this.favoriteService.load(query);
  }
}

```

The `TopStoriesPage` is already simplified by inheriting from `AbstractFavoriteItemsPage`. The implementation of `getRawItems()` returns the value from the `ItemService`. See Listing 8-4.

Listing 8-4. *TopStoriesPage*

```

import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { LoadingController } from 'ionic-angular';
import { ToastController } from 'ionic-angular';
import { Items } from '../..model/Items';

```

```
import { ItemService } from '../../../services/ItemService';
import { Observable } from "rxjs";
import 'rxjs/add/operator/combineLatest';
import { OpenPageService } from "../../../services/OpenPageService";
import { Query } from '../../../services/AbstractItemService';
import { FavoriteService } from '../../../services/FavoriteService';
import { AuthService } from '../../../services/AuthService';
import { AbstractFavoriteItemsPage } from '../AbstractFavoriteItemsPage';

@Component({
  selector: 'page-top-stories',
  templateUrl: 'top-stories.html'
})
export class TopStoriesPage extends AbstractFavoriteItemsPage {
  constructor(public navCtrl: NavController,
    public loadingCtrl: LoadingController,
    public toastCtrl: ToastController,
    protected itemService: ItemService,
    protected openPageService: OpenPageService,
    protected authService: AuthService,
    protected favoriteService: FavoriteService) {
    super(navCtrl, loadingCtrl, toastCtrl, openPageService, authService,
      favoriteService);
  }

  ngOnInit(): void {
    super.ngOnInit();
  }

  ngOnDestroy(): void {
    super.ngOnDestroy();
  }

  getRawItems(): Observable<Items> {
    return this.itemService.get();
  }

  query(query: Query): void {
    this.itemService.load(query);
  }
}
```

Favorites Items

Once the favorites service and page are done, we need to update the `ItemComponent` to show the buttons for toggling favorites state. To accomplish this, we need to pass the `enableFavorite` flag and method

references `addToFavorite` and `removeFromFavorite` to the `ItemComponent`. We follow the same pattern as we did for the `openPage` and `viewComment` references. We declare two new function types in model `Item`. Both `AddToFavorite` and `RemoveFromFavorite` accept the item id as the parameter and return a `Promise<void>` object; see Listing 8-5.

Listing 8-5. AddToFavorite and RemoveFromFavorite

```
export interface AddToFavorite {
  (itemId: number): Promise<void>;
}

export interface RemoveFromFavorite {
  (itemId: number): Promise<void>;
}
```

Then we declare two `@Input()` properties in `ItemComponent` and `ItemsComponent` for both `AddToFavorite` and `RemoveFromFavorite`. Properties `addToFavoriteDisabled` and `removeFromFavoriteDisabled` are used to control the disabled state of buttons. In the method `add()`, we first disable the button by setting `addToFavoriteDisabled` to true, then invoke the method `addToFavorite()`. The button is enabled after the operation finishes; see Listing 8-6.

Listing 8-6. Updated ItemComponent

```
export class ItemComponent {
  @Input() item: Item;
  @Input() enableFavorite: boolean;
  addToFavoriteDisabled: boolean = false;
  @Input() addToFavorite: AddToFavorite;
  removeFromFavoriteDisabled: boolean = false;
  @Input() removeFromFavorite: RemoveFromFavorite;

  add() {
    this.addToFavoriteDisabled = true;
    this.addToFavorite(this.item.id)
      .then(_ => this.addToFavoriteDisabled = false)
      .catch(_ => this.addToFavoriteDisabled = false);
  }

  remove() {
    this.removeFromFavoriteDisabled = true;
    this.removeFromFavorite(this.item.id)
      .then(_ => this.removeFromFavoriteDisabled = false)
      .catch(_ => this.removeFromFavoriteDisabled = false);
  }
}
```

In the template of `ItemComponent` in Listing 8-7, the directive `ngIf` makes sure these buttons are only displayed when `enableFavorite` is true. It depends on whether the item is in the favorites, that is, `isFavorite` flag, we show different buttons with different actions; see Listing 8-7.

Listing 8-7. Update template of `ItemComponent`

```
<span *ngIf="enableFavorite">
  <button class="btnLike" ion-button clear full icon-left *ngIf="!item.
    isFavorite" [disabled]="addToFavoriteDisabled" (click)="add()">
    <ion-icon name="heart-outline" color="danger"></ion-icon>
    Like
  </button>
  <button class="btnUnlike" ion-button clear full icon-left *ngIf="item.
    isFavorite" [disabled]="removeFromFavoriteDisabled" (click)="remove()">
    <ion-icon name="heart" color="danger"></ion-icon>
    Liked
  </button>
</span>
```

We also need to modify the template of `ItemsComponent` to pass those three properties down to the `ItemComponent`. In the template of `TopStoriesPage` and `FavoritesPage`, we pass those properties to the `ItemsComponent`. Because the type of `enableFavorite` in `TopStoriesPage` is `Observable<boolean>`, we need to use the `async` pipe to extract the value before passing it; see Listing 8-8.

Listing 8-8. Updated template of `ItemsComponent`

```
<items *ngIf="items"
  [items]="items"
  [openPage]="openPage"
  [viewComment]="viewComment"
  [enableFavorite]="enableFavorite | async"
  [addToFavorite]="addToFavorite"
  [removeFromFavorite]="removeFromFavorite">
</items>
```

Figure 8-1 shows the screenshot of the top stories page after adding the favorites.

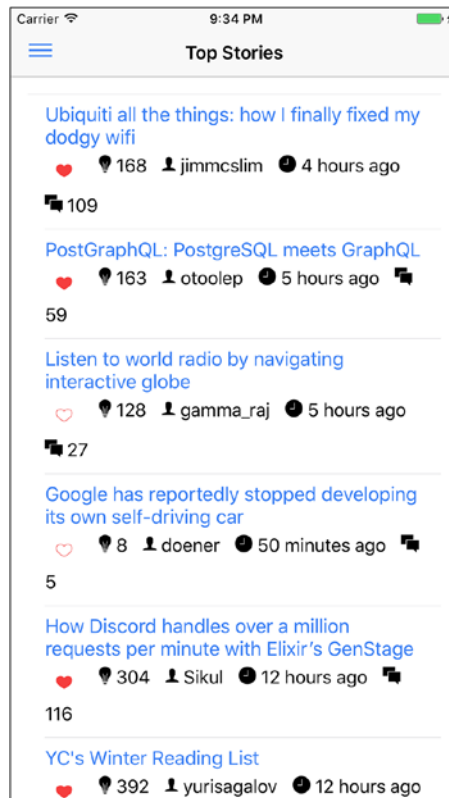


Figure 8-1. Screenshot of top stories page with favorites

Testing

We are going to add test specs for the favorites page. Before we can test the favorites page, we need to create a mock for the `FavoriteService`. The `FavoriteServiceMock` in Listing 8-9 extends from the same parent class `AbstractItemService`, but it uses an in-memory object store to hold all favorite items.

In the `FavoriteServiceMock`, the store is the map from item id to the `BehaviorSubject` object, `itemIds` is the array of ids of favorite items, `items` is the `BehaviorSubject` object that emits ids of favorite items. In the method `add()`, if the item is not in the favorites, a new `BehaviorSubject` object is created and added to the store. Otherwise, we update the existing `BehaviorSubject` object with the current timestamp. We also update the array `itemIds` to add this new item. The `items` also emits the updated value of `itemIds`. In the method `remove()`, we emit a null value from the `BehaviorSubject` object for this favorite item. `itemIds` is also updated and

items also emits the updated value of itemIds. The method getItemIds() simply returns the items object. The method getFavoriteItem() returns the saved BehaviorSubject objects in the store or a new BehaviorSubject object if it doesn't exist in the store.

Listing 8-9. FavoriteServiceMock

```
import { Injectable } from '@angular/core';
import { Observable, BehaviorSubject } from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/mergeMap';
import { AuthService } from '../services/AuthService';
import { HackerNewsServiceMock } from '../HackerNewsServiceMock';
import { AbstractItemService } from '../services/AbstractItemService';
import * as pull from 'lodash.pull';

type Store = { [key:number]: BehaviorSubject<any>; };

@Injectable()
export class FavoriteServiceMock extends AbstractItemService {
  store: Store = {};
  itemIds: number[] = [];
  items: BehaviorSubject<number[]> = new BehaviorSubject<number[]>([]);
  constructor(private authService: AuthService) {
    super(new HackerNewsServiceMock() as any);
  }

  add(itemId: number) {
    const user = this.authService.user.getValue();
    if (user == null) {
      return;
    }
    const value = {
      $value: new Date().getTime(),
    };
    if (this.store[itemId]) {
      this.store[itemId].next(value);
    } else {
      this.store[itemId] = new BehaviorSubject(value);
    }
    this.itemIds.unshift(itemId);
    this.items.next(this.itemIds);
  }

  remove(itemId: number) {
    const user = this.authService.user.getValue();
    if (user == null) {
      return;
    }
  }
```

```

    const value = {
      $value: null,
    };
    if (this.store[itemId]) {
      this.store[itemId].next(value);
    }
    pull(this.itemIds, itemId);
    this.items.next(this.itemIds);
  }

  getItemIds(): Observable<number[]> {
    return this.items;
  }

  getFavoriteItem(itemId: number): Observable<any> {
    return this.authService.user.mergeMap(user => {
      if (user) {
        if (!this.store[itemId]) {
          this.store[itemId] = new BehaviorSubject({
            $value: null,
          });
        }
        return this.store[itemId];
      } else {
        return Observable.of({
          $value: null,
        });
      }
    });
  }
}

```

Testing favorites requires the user to log in first; we use the `AuthServiceMock` in Listing 8-10 as the mock object for `AuthService`. In the method `login()` of `AuthServiceMock`, the passed-in user object is used directly without any checks.

Listing 8-10. *AuthServiceMock*

```

import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
import { User } from '../model/User';

@Injectable()
export class AuthServiceMock {
  user: BehaviorSubject<User>;
  constructor() {
    this.user = new BehaviorSubject<User>(null);
  }
}

```

```
create(user: User) {
  this.logIn(user);
  return Promise.resolve(user);
}

logIn(user: User) {
  this.user.next(user);
}

logOut() {
  this.user.next(null);
}
}
```

For the test spec of FavoritesPage, we set up the testing bed with the same mock objects as the test spec for TopStoriesPage. Since these mock objects are used in both pages, we refactor these common mocks into a separate file in Listing 8-11. The method createCommonMocks() returns new objects of common mocks.

Listing 8-11. Common mocks

```
export function createCommonMocks() {
  const loadingStub = jasmine.createSpyObj('loading', ['present',
    'dismiss']);
  const toastStub = jasmine.createSpyObj('toast', ['present']);
  const alertStub = jasmine.createSpyObj('alert', ['present', 'dismiss']);
  const modalStub = jasmine.createSpyObj('modal', ['present', 'dismiss',
    'onDidDismiss']);
  const openPageServiceStub = jasmine.createSpyObj('openPage', ['open']);
  return {
    loadingControllerStub: {
      create: jasmine.createSpy('create loading').and.returnValue(loadingStub),
    },
    toastControllerStub: {
      create: jasmine.createSpy('create toast').and.returnValue(toastStub),
    },
    alertControllerStub: {
      create: jasmine.createSpy('create alert').and.returnValue(alertStub),
    },
    viewControllerStub: Object.assign(jasmine.createSpyObj('view',
      ['dismiss', '_setHeader', '_setIONContent', '_setIONContentRef']), {
      readReady: new Subject(),
      writeReady: new Subject(),
    }),
  },
}
```

```

    modalControllerStub: {
      create: jasmine.createSpy('create modal').and.returnValue(modalStub),
    },
    openPageServiceStub,
  };
}

```

In the first test spec, we use `AuthService` to log in a test user first, then add three items to the favorites, and verify that three items are displayed in the favorites page. In the second test spec, we first add three items to the favorites, then remove the first item. After the removal, there should be only two items with the liked state. After that we do a refresh, and there should be only two items in the favorites list; see Listing 8-12.

Listing 8-12. FavoritesPage test suite

```

import { ComponentFixture, async } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { TestUtils } from '../test';
import { LoadingController, ToastController, AlertController } from
'ionic-angular';
import { FavoritesPage } from './favorites';
import { User } from '../model/User';
import { ItemsComponent } from '../components/items/items';
import { ItemComponent } from '../components/item/item';
import { TimeAgoPipe } from '../pipes/TimeAgoPipe';
import { FavoriteServiceMock } from '../testing/FavoriteServiceMock';
import { FavoriteService } from '../services/FavoriteService';
import { AuthService } from '../services/AuthService';
import { AuthServiceMock } from '../testing/AuthServiceMock';
import { OpenPageService } from '../services/OpenPageService';
import { createCommonMocks } from '../testing/CommonMock';

let fixture: ComponentFixture<FavoritesPage> = null;
let component: any = null;

describe('favorites page', () => {
  beforeEach(async(() => {
    const {
      loadingControllerStub,
      toastControllerStub,
      alertControllerStub,
      openPageServiceStub,
    } = createCommonMocks();

```

```
TestUtils.beforeEachCompiler(
  [FavoritesPage, ItemsComponent, ItemComponent, TimeAgoPipe],
  [
    {provide: FavoriteService, useClass: FavoriteServiceMock},
    {provide: LoadingController, useValue: loadingControllerStub},
    {provide: ToastController, useValue: toastControllerStub},
    {provide: AlertController, useValue: alertControllerStub},
    {provide: OpenPageService, useValue: openPageServiceStub},
    {provide: AuthService, useClass: AuthServiceMock},
  ]
).then(compiled => {
  fixture = compiled.fixture;
  component = compiled.instance;
});
));

it('should display favorites items', async(() => {
  const authService = fixture.debugElement.injector.get(AuthService);
  const favoriteService = fixture.debugElement.injector.
    get(FavoriteService);
  authService.login(new User('user', 'user', 'user@test.com', ''));
  favoriteService.add(1);
  favoriteService.add(2);
  favoriteService.add(3);
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    let debugElements = fixture.debugElement.queryAll(By.css('ion-item'));
    expect(debugElements.length).toBe(3);
  });
});

it('should handle removal of favorites items', async(() => {
  const authService = fixture.debugElement.injector.get(AuthService);
  const favoriteService = fixture.debugElement.injector.
    get(FavoriteService);
  authService.login(new User('user', 'user', 'user@test.com', ''));
  favoriteService.add(1);
  favoriteService.add(2);
  favoriteService.add(3);
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    let debugElements = fixture.debugElement.queryAll(By.css
      ('ion-icon[name=heart]'));
    expect(debugElements.length).toBe(3);
    favoriteService.remove(1);
    fixture.detectChanges();
  });
});
```

```

    fixture.whenStable().then(() => {
      let debugElements = fixture.debugElement.queryAll(By.css
        ('ion-icon[name=heart]'));
      expect(debugElements.length).toBe(2);
      component.doRefresh();
      fixture.detectChanges();
      fixture.whenStable().then(() => {
        let debugElements = fixture.debugElement.queryAll(By.css
          ('ion-item'));
        expect(debugElements.length).toBe(2);
      });
    });
  });
}));
});

```

Summary

In this chapter, we implement the user story to manage favorites, including adding stories into the favorites and removing stories from the favorites. Favorites data is stored in Firebase. We discuss different data storage formats and implement the service to interact with Firebase. We also create the page to list stories in the favorites. Since the favorites feature is only enabled when the user is logged in, we also discuss the integration with the AuthService created in the last chapter. In the next chapter, we'll implement the user story to share stories.

Share Stories

Now we move to the last user story on the list that allows users to share stories.

Card Layout

To allow users to share stories, we need to add a new button. If we add this button to the current UI, then all these UI components cannot fit into one line. We need a new layout for all those components. A good choice is the card layout.

Cards are created using the component `ion-card`. A card can have a header and content that can be created using `ion-card-header` and `ion-card-content`, respectively. Listing 9-1 shows a simple card with a header and content.

Listing 9-1. A simple card

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

In the `ion-card-content`, we can include different kinds of components. The component `ion-card-title` can be used to add title text to the content. Listing 9-2 shows a card with an image and a title.

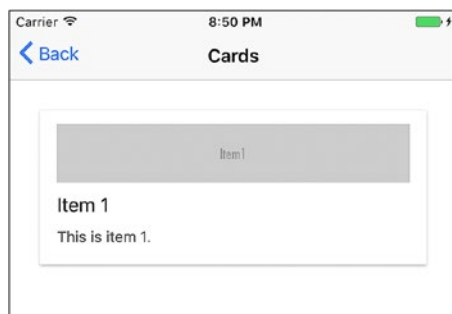
Listing 9-2. A complex card

```

<ion-card>
  <ion-card-content>
    
    <ion-card-title>Item 1</ion-card-title>
    <p>
      This is item 1.
    </p>
  </ion-card-content>
</ion-card>

```

See Figure 9-1 for the screenshot of the card in Listing 9-2.

**Figure 9-1.** Card

After using the card layout, each item in the list is rendered as a card.

Grid Layout

In the item card, we need to display two or three buttons. These buttons should take up the same horizontal space of the line. This layout requirement can be easily archived by using the grid layout. The grid layout is implemented using CSS3 flexbox layout.

The grid layout uses three components, `ion-grid`, `ion-row`, and `ion-col`. `ion-grid` represents the grid itself, `ion-row` represents a row in the grid, `ion-col` represents a column in a row. Rows take up the full horizontal space in the grid and flow from top to bottom. Horizontal space of a row

is distributed evenly across all columns in the row. We can also specify the width percentage for each column using properties `width-10`, `width-20`, `width-25`, `width-33`, `width-50`, `width-67`, `width-75`, `width-80` and `width-90`. The number after `width-` is the width percentage, for example, `width-25` means 25%. By default, columns in a row flow from left to right and are placed next to each other. We can use the property `offset` to specify the offset from the left side. We can use the same percentage values of `width` in `offset`, e.g. `offset-25`, `offset-75`.

For the vertical align on each row, we can use `center` or `baseline` on `ion-row`. `ion-row` also supports using the property `wrap` to allow items in the row to wrap.

Listing 9-3 shows an example of using the grid layout to create the basic UI of a calculator.

Listing 9-3. Grid layout

```
<ion-grid>
  <ion-row>
    <ion-col><button ion-button full>1</button></ion-col>
    <ion-col><button ion-button full>2</button></ion-col>
    <ion-col><button ion-button full>3</button></ion-col>
  </ion-row>
  <ion-row>
    <ion-col><button ion-button full>4</button></ion-col>
    <ion-col><button ion-button full>5</button></ion-col>
    <ion-col><button ion-button full>6</button></ion-col>
  </ion-row>
  <ion-row>
    <ion-col><button ion-button full>7</button></ion-col>
    <ion-col><button ion-button full>8</button></ion-col>
    <ion-col><button ion-button full>9</button></ion-col>
  </ion-row>
  <ion-row>
    <ion-col width-33><button ion-button full>0</button></ion-col>
    <ion-col width-67><button ion-button full color="secondary">=</button>
  </ion-col>
</ion-row>
</ion-grid>
```

Figure 9-2 shows the screenshot of Listing 9-3.

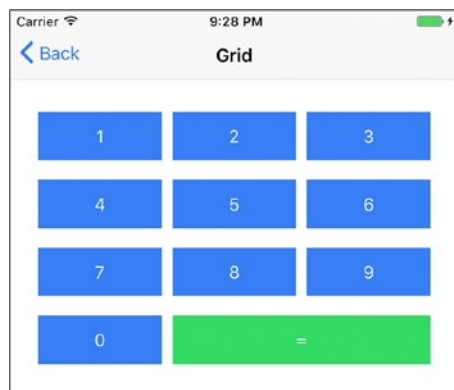


Figure 9-2. A calculator UI using grid layout

Sharing

We can use the Cordova plugin Social Sharing (<https://github.com/EddyVerbruggen/SocialSharing-PhoneGap-Plugin>) to share stories via different channels. We need to install the plugin cordova-plugin-x-socialsharing using Ionic CLI.

```
$ ionic plugin add cordova-plugin-x-socialsharing --save
```

We create the new `SocialSharingService` for the social sharing functionality. It has a single method `share(message: string, url: string)` with the message for the sharing and the url of the shared item. It simply calls the method `share()` from `SocialSharing` of the Ionic Native wrapper of this Cordova plugin. It pops up the platform specific sharing sheet to do the sharing. See Listing 9-4.

Listing 9-4. *SocialSharingService*

```
import { Injectable } from '@angular/core';
import { SocialSharing } from 'ionic-native';

@Injectable()
export class SocialSharingService {
  share(message: string, url: string) {
    SocialSharing.share(message, null, null, url);
  }
}
```

Then we update the template of `ItemComponent` to include a new button to do the sharing. We pass the item's title as the message to display. In the card for each item, we use three rows in the grid. The first row contains the item's title. The second row contains the item's score, author, and published time. The last row contains buttons for toggling favorites, sharing and viewing comments; see Listing 9-5.

Listing 9-5. Updated ItemComponent

```
<ion-card *ngIf="!item">
  <ion-card-header>
    Loading...
  </ion-card-header>
</ion-card>
<ion-card *ngIf="item">
  <ion-grid>
    <ion-row>
      <ion-col>
        <ion-item text-wrap (click)="openPage(item.url)">
          <h2 class="title">{{ item.title }}</h2>
        </ion-item>
      </ion-col>
    </ion-row>
    <ion-row>
      <ion-col padding-left>

        </ion-col>
    </ion-row>
    <ion-row>
      <ion-col *ngIf="enableFavorite">

        </ion-col>
      <ion-col>
        <button ion-button clear full icon-left (click)="share
          (item.title, item.url)">
          <ion-icon name="share"></ion-icon>
          Share
        </button>
      </ion-col>
      <ion-col *ngIf="item.descendants" (click)="viewComment(item.id)">

        </ion-col>
    </ion-row>
  </ion-grid>
</ion-card>
</ion-col>
```

```
<button ion-button clear full icon-left (click)="share
(item.title, item.url)">
  <ion-icon name="share"></ion-icon>
  Share
</button>
</ion-col>
```

The method `share()` is added to class `AbstractFavoriteItemsPage`, so both `TopStoriesPage` and `FavoritesPage` can use this method.

More About the Plugin

The method `share(message, subject, file, url)` of `SocialSharing` actual takes four arguments:

- `message` – The message to share with.
- `subject` – The subject of the sharing.
- `file` – URL or local path to files or images. Can be an array for sharing multiple files or images.
- `url` – The URL to share.

The method `share()` uses the sharing sheet to let the user choose the sharing channels. We can also use other methods to directly share with specific social providers.

- `shareViaTwitter(message, image, url)` – Share to Twitter.
- `shareViaFacebook(message, image, url)` – Share to Facebook.
- `shareViaInstagram(message, image)` – Share to Instagram.
- `shareViaWhatsApp(message, image, url)` – Share to WhatsApp.
- `shareViaSMS(message, phoneNumber)` – Share via SMS.
- `shareViaEmail(message, subject, to, cc, bcc, files)` – Share via emails.

Or we can use the generic method `shareVia(appName, message, subject, image, url)` to share via any app. Because the sharing capability depends on the native apps installed on the user's device, it's better to check whether the sharing is possible first. The method `canShareVia(appName, message, subject, image, url)` does the check for a given app. For sharing via emails, the specific method `canShareViaEmail()` should be used for the check.

Summary

In this chapter, we implement the user story to share Hacker News stories to social networking services. Since we need to add a new button for each story in the list to trigger the sharing action, we also use the Ionic 2 card and grid layout to make the UI look better. In the next chapter, we'll discuss some common Ionic 2 components that are not used in the example app.

Common Components

When implementing these user stories for the app, we already use many Ionic built-in components. There are still some Ionic components that are useful but not used in the app. We are going to discuss several important components.

Action Sheet

An action sheet is a special kind of dialog that lets a user choose from a group of options. It's like the alert component we mentioned before, but only buttons are allowed in an action sheet. It can also be used as menus. An action sheet contains an array of buttons. There are three kinds of buttons in action sheets: destructive, normal, or cancel. This can be configured by setting the property `role` to `destructive` or `cancel`. Destructive buttons usually represent dangerous actions, for example, deleting an item or canceling a request. Destructive buttons have different styles to clearly convey the message to the user and they usually appear first in the buttons array. Cancel buttons always appear last in the buttons array.

Action sheets are created using the method `create()` of `ActionSheetController`. The method `create()` takes an options object with the following possible properties.

- `title` – The title of the action sheet.
- `subTitle` – The subtitle of the action sheet.
- `cssClass` – The extra CSS classes to add to the action sheet.

- `enableBackdropDismiss` – Whether the action sheet should be dismissed when the backdrop is tapped.
- `buttons` – The array of buttons to display in the action sheet.

Each button in the of array of buttons is a JavaScript object with the following possible properties.

- `text` – The text of the button.
- `icon` – The icon of the button.
- `handler` – The handler function to invoke when the button is pressed.
- `cssClass` – The extra CSS classes to add to the button.
- `role` – The role of the button. Possible values are `destructive` and `cancel`.

The return value of `create()` is an `ActionSheet` instance, and we can use methods `present()` or `dismiss()` to present or dismiss the action sheet, respectively. When the action sheet is dismissed by the user tapping the backdrop, the handler of the button with the role `cancel` is invoked automatically.

Listing 10-1 shows an action sheet with three buttons.

Listing 10-1. Action sheet

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

@Component({
  selector: 'page-actionsheet',
  templateUrl: 'actionsheet.html'
})
export class ActionsheetPage {
  actionSheet: ActionSheet;
  constructor(public navCtrl: NavController,
               public actionSheetCtrl: ActionSheetController) {}

  fileAction() {
    this.actionSheet = this.actionSheetCtrl.create({
      title: 'Choose your action',
      enableBackdropDismiss: true,
      buttons: [
        {
          text: 'Delete',
          role: 'destructive',
```



```

        icon: 'trash',
        handler: this.deleteFile.bind(this),
    },
    {
        text: 'Move',
        icon: 'move',
        handler: this.moveFile.bind(this),
    },
    {
        text: 'Cancel',
        role: 'cancel',
        icon: 'close',
        handler: this.close.bind(this),
    }
]
});
this.actionSheet.present();
}

close() {
    this.actionSheet.dismiss();
}

deleteFile() {
}

moveFile() {
}
}

```

Figure 10-1 shows the screenshot of the action sheet.

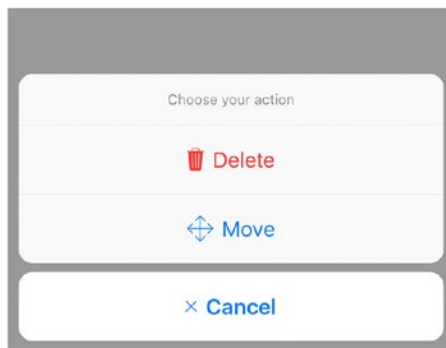


Figure 10-1. Action sheet

Popover

A popover is like the modal dialog we mentioned before, but it's not modal. The popover floats on top of the current page. Like modals, popovers are created by wrapping existing components. Popovers are created with the method `create()` of `PopoverController`. The method `create(component, data, opts)` has three parameters.

- `component` – The component that's wrapped in the popover.
- `data` – The data object to pass to the popover component.
- `opts` – The options of creating the popover.

The options object supports properties `cssClass`, `showBackdrop` and `enableBackdropDismiss` with the same meanings as in other components.

The return value of `create()` is a `Popover` instance. The popover can be dismissed by invoking `dismiss()` of the `Popover` instance. The popover can also be dismissed within the wrapped component by invoking `dismiss()` on the `ViewController` instance. The method `dismiss()` can accept an optional object that passed to the callback function configured by `onDidDismiss()` of the `Popover` instance. This is how data is passed between the component wrapped by the popover and the component that creates the popover.

Now we use an example to demonstrate how to pass data when using popovers; see Listing 10-2. The component contains some text and we use a popover to change the font size. In the `PopoverPage`, we use the injected `PopoverController` instance to create a new `Popover`. When invoking `create()`, the component to show is `FontSizeChooserComponent` and we pass the current value of `fontSize` to the component. We use `present()` to show the popover. The event object of the click event is passed as the argument, so the popover is positioned based on the position of the click event. If no event is passed, the popover will be positioned in the center of current view. We then use `onDidDismiss()` to add a callback function to receive the updated value of `fontSize` from the popover.

Listing 10-2. PopoverPage

```
import { Component } from '@angular/core';
import { NavController, PopoverController } from 'ionic-angular';
import { FontSizeChooserComponent } from '../components/font-size-
chooser/font-size-chooser';

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

```

export class PopoverPage {
  fontSize: number = 8;
  constructor(public navCtrl: NavController,
               public popoverCtrl: PopoverController) {}

  changeFontSize(event) {
    const popover = this.popoverCtrl.create(FontSizeChooserComponent, {
      fontSize: this.fontSize
    });
    popover.present({
      ev: event
    });
    popover.onDidDismiss((fontSize) => {
      this.fontSize = fontSize;
    });
  }
}

```

The template of the PopoverPage is very simple. It has a button to create the popover. The font size of the <div> element is updated using the directive `ngStyle`. See Listing 10-3.

Listing 10-3. Template of PopoverPage

```

<ion-content padding>
  <div [ngStyle]="{'font-size': fontSize + 'px'}">
    Hello world
  </div>
  <button ion-button (click)="changeFontSize($event)">Font size</button>
</ion-content>

```

In the FontSizeChooserComponent, when the method `save()` is called, the injected `ViewController` instance is used to dismiss the popover and pass the current value of `fontSize`. `NavParams` is used to get the value of `fontSize` that passed in when the popover is presented. See Listing 10-4.

Listing 10-4. FontSizeChooserComponent

```

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

@Component({
  selector: 'font-size-chooser',
  templateUrl: 'font-size-chooser.html'
})
export class FontSizeChooserComponent {
  fontSize: number;

```

```

constructor(private viewCtrl: ViewController,
             private navParams: NavParams) {
    this.fontSize = navParams.data.fontSize;
}

save() {
    this.viewCtrl.dismiss(this.fontSize);
}
}

```

In the template of `FontSizeChooserComponent`, we use an `ion-range` component to let the user select the font size. See Listing 10-5.

Listing 10-5. Template of `FontSizeChooserComponent`

```

<ion-list>
  <ion-list-header>Font size</ion-list-header>
  <ion-item>
    <ion-range [(ngModel)]="fontSize" pin="true" min="8" max="32">
      <ion-icon range-left name="remove"></ion-icon>
      <ion-icon range-right name="add"></ion-icon>
    </ion-range>
  </ion-item>
  <ion-item>
    <button ion-button full (click)="save()">OK</button>
  </ion-item>
</ion-list>

```

Figure 10-2 shows the screenshot of the font size chooser.

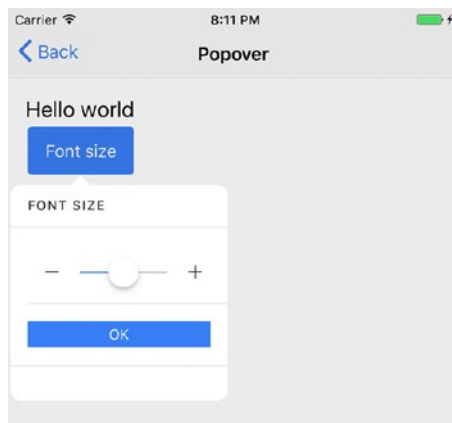


Figure 10-2. Font size chooser

Slides

The slides component is a container for multiple views. The user can swipe or drag between different views. Slides are commonly used for tutorials and galleries.

Slides are created using components `ion-slides` and `ion-slide`. `ion-slides` is the container component for `ion-slide` components inside of it. When creating the `ion-slides`, we can use the property options to configure it.

- `autoplay` – The interval (in milliseconds) between each view transition.
- `direction` – The direction to swipe. Possible values are `horizontal` and `vertical`. The default value is `horizontal`.
- `initialSlide` – The index of the initial slide. The default value is 0.
- `loop` – Whether to loop back from the last slide to the first slide. The default value is `false`.
- `pager` – Whether to show the pagination bullets. The default value is `false`.
- `speed` – The duration of transition between slides in milliseconds. The default value is 300.

After the slides component is created, we can also programmatically control the slide transitions using the following methods.

- `slideTo(index, speed, runCallbacks)` – Transition to the slide with specified index. `speed` is the transition duration. When `runCallbacks` is `true`, `ionWillChange` and `ionDidChange` events will be emitted.
- `slideNext(speed, runCallbacks)` – Transition to the next slide.
- `slidePrev(speed, runCallbacks)` – Transition to the previous slide.

In Listing 10-6, we create an `ion-slides` component with the reference variable set to `slides`. It contains three `ion-slide` components. `slides-control` is a component to control the slides.

Listing 10-6. Slides

```
<ion-slides #slides>
  <ion-slide>
    Slide 1
    <slides-control [slides]="this.slides"></slides-control>
  </ion-slide>
  <ion-slide>
    Slide 2
    <slides-control [slides]="this.slides"></slides-control>
  </ion-slide>
  <ion-slide>
    Slide 3
    <slides-control [slides]="this.slides"></slides-control>
  </ion-slide>
</ion-slides>
```

Listing 10-7 is the template of component `slides-control`. It has two buttons to go to the previous or next slide.

Listing 10-7. Template of slides-control

```
<div>
  <button ion-button (click)="prev()" [disabled]="!hasPrev()">Prev</button>
  <button ion-button (click)="next()" [disabled]="!hasNext()">Next</button>
</div>
```

The component `SlidesControl` in Listing 10-8 has an `@Input` property `slides` that binds to a `Slides` object. The method `isValid()` is required to check whether the `Slides` component is ready to use.

Listing 10-8. SlidesControl and SlidesPage

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

@Component({
  selector: 'slides-control',
  templateUrl: 'slides-control.html'
})
export class SlidesControl {
  @Input() slides: Slides;

  prev() {
    if (this.isValid()) {
      this.slides.slidePrev();
    }
  }
}
```

```

next() {
    if (this.isValid()) {
        this.slides.slideNext();
    }
}

hasPrev() {
    return this.isValid() && !this.slides.isBeginning();
}

hasNext() {
    return this.isValid() && !this.slides.isEnd();
}

isValid(): boolean {
    return !(this.slides && this.slides.slider);
}
}

@Component({
    selector: 'page-slides',
    templateUrl: 'slides.html'
})
export class SlidesPage {
    @ViewChild('slides') slides: Slides;
    constructor(public navCtrl: NavController) {}
}

```

Figure 10-3 shows the screenshot of the slides component.



Figure 10-3. *Slides*

Tabs

Tabs are commonly used components for layout and navigation. Different tabs can take the same screen estate, and only one tab can be active at the same time.

Tabs components are created using the component `ion-tabs`, while individual tabs are created using `ion-tab`. `ion-tabs` supports the standard properties `color` and `mode` and following special properties.

- `selectedIndex` – The default selected index of the tab. The default value is 0.
- `tabsLayout` – The layout of the tab bar. Possible values are `icon-top`, `icon-left`, `icon-right`, `icon-bottom`, `icon-hide`, `title-hide`.
- `tabsPlacement` – The position of the tab bar. Possible values are `top` and `bottom`.
- `tabsHighlight` – Whether to show a highlight bar under the selected tab. The default value is `false`.

Once `ion-tabs` is created, we can get the Tabs instance of this component. Tabs instance provides different methods to interact with the tabs.

- `select(tabOrIndex)` – Select a tab by its index or its Tab instance.
- `previousTab(trimHistory)` – Get the previously selected tab that is not disabled or hidden.
- `getByIndex(index)` – Get the Tab instance by index.
- `getSelected()` – Get the selected Tab instance.

Each `ion-tab` is a `NavController`, so it has its own navigation stack. It also supports the following properties to configure it.

- `root` – The root page for this tab to show.
- `rootParams` – `NavParams` passed to the root page of this tab.
- `tabTitle` – The title of the tab.
- `tabIcon` – The icon of the tab.
- `tabBadge` – The badge to display on the tab button.
- `tabBadgeStyle` – The color of the badge.
- `enabled` – Whether the tab button is enabled.

- show – Whether the tab button is visible.
- swipeBackEnabled – Whether the user can swipe to go back to previous tab.
- tabsHideOnSubPages – Whether the tab is hidden on sub pages.

ion-tab also emits the event ionSelect when it's selected.

In the template file of Listing 10-9, we create an ion-tabs with reference name set to sampleTabs. Each ion-tab has its title and icon. The second tab uses rootParams to pass an object to its root page.

Listing 10-9. Template of tabs

```
<ion-tabs #sampleTabs color="primary">
  <ion-tab [root]="tab1Root" tabTitle="Tab 1" tabIcon="alarm"></ion-tab>
  <ion-tab [root]="tab2Root" [rootParams]="{value: 'Test'}" tabTitle="Tab 2"
    tabIcon="albums"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Tab 3" tabIcon="settings"></ion-tab>
</ion-tabs>
```

In the TabsPage of Listing 10-10, we use the decorator @ViewChild to get the reference to Tabs object and use its method select() to select the second tab. In the first tab page TabPage1, by using the property parent of the injected NavController instance, we can get the reference to the Tabs object. In the TabPage2, we get the data passed in from Tabs to this page using NavParams.

Listing 10-10. Tabs

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

@Component({
  selector: 'page-tabs',
  templateUrl: 'tabs.html'
})
export class TabsPage {
  tab1Root = TabPage1;
  tab2Root = TabPage2;
  tab3Root = TabPage3;
  @ViewChild('sampleTabs') tabsRef: Tabs;
  constructor(public navCtrl: NavController) {}

  ionViewDidEnter() {
    this.tabsRef.select(1);
  }
}
```

```
@Component({
  template: '<p><button ion-button (click)="selectPage()">Select Page 3
</button></p>',
})
export class TabPage1 {
  constructor(public navCtrl: NavController) {

  }

  selectPage() {
    this.navCtrl.parent.select(2);
  }
}

@Component({
  template: '<p>Value: {{value}}</p>',
})
export class TabPage2 {
  value: string;
  constructor(private navParams: NavParams) {
    this.value = navParams.data.value;
  }
}

@Component({
  template: '<p>Page 3</p>',
})
export class TabPage3 {

}
```

Summary

In this chapter, we discuss how to use other common Ionic 2 components that are not used in the example app, including action sheet, popover, slides, and tabs. For each component, we provide a concrete example to demonstrate the usage. In the next chapter, we'll discuss some advanced topics in Ionic 2.

Advanced Topics

In this chapter, we discuss some advanced topics.

Platform

We have seen the class `Platform` in previous chapters. It can be used to interact with the underlying platform. We used the method `ready()` of `Platform` to wait for the Cordova platform to finish initialization. The class `Platform` also has other methods.

- `platforms()` – Depends on the running device, the return value can be an array of different platforms. Possible values of `platforms` are `android`, `cordova`, `core`, `ios`, `ipad`, `iphone`, `mobile`, `mobileweb`, `phablet`, `tablet`, and `windows`. When running on the iPhone emulator, the return value of `platforms()` is `["cordova", "mobile", "ios", "iphone", "phablet"]`.
- `is(platformName)` – Check if the running platform matches the given platform name. Because the method `platforms()` can return multiple values, the method `is()` can return `true` for multiple values.
- `versions()` – Get version information for all the platforms. When running on the iPhone emulator, the return value of `versions()` is `{"ios":{"str":"10.2", "num":10.2, "major":10, "minor":2}}`.
- `dir()` – Get the language direction. Possible values are `ltr` and `rtl`. In the `index.html`, the attribute `dir` of the element `<html>` should be used to set the language direction.

- `isRTL()` – Check if the language direction is right to left.
- `setDir(dir)` – Set the language direction. Possible values of the parameter `dir` can be `ltr` and `rtl`.
- `lang()` – Get the language and an optional country code. In the `index.html`, the attribute `lang` of the element `<html>` should be used to set the language.
- `setLang(language)` – Set the language and an optional country code. Possible values are like `en`, `en-US` or `zh-CN`.
- `width()` and `height()` – Get the width and height of the platform's viewport, respectively.
- `isPortrait()` and `isLandscape()` – Check if the app is in portrait or landscape mode, respectively.
- `registerBackButtonAction(callback, priority)` – Android and Windows platforms have the hardware back button. This method can be used to add event handlers when this hardware back button is pressed. The first parameter `callback` is the function to invoke. The second parameter is the priority of this event handler. Only the event handler with the highest priority will be called.
- `ready()` – This method returns a promise that resolved when the platform is ready and we can use the native functionalities. The resolved value is the name of the platform that was ready.

There are two important `EventEmitters` in the `Platform` that are related to app states. The `EventEmitter` `pause` emits events when the app is put into background. The `EventEmitter` `resume` emits events when the app is pulled out from the background. These two `EventEmitters` are useful when dealing with app state changes.

Theming

Ionic provides different look and feels based on the current platform. The styles are grouped as different modes. Each platform has a default mode that can also be overridden. It's possible to use iOS styles on Android devices.

There are three modes: `md` for Material Design styles, `ios` for iOS styles, `wp` for the Windows styles. The platform `ios` uses the `ios` mode by default, `android` uses the `md` mode, `windows` uses the `wp` mode. Other platforms use the `md` mode.

Once the mode is selected for the app, the element `<ion-app>` will have the mode name as a CSS class name, for example, `<ion-app class="md">` for the mode `md`. This class name can be used to override styles for different modes. In the code below, we add extra styles only for the mode `md`.

```
.md {
  font-size: 16px;
}
```

Ionic components have the property `mode` to set the mode. This mode overrides the platform's default mode for this component only. In Listing 11-1, we declare three `ion-button` components with different modes.

Listing 11-1. Different modes for ion-button components

```
<div>
  <button ion-button full mode="md">Material Design</button>
  <button ion-button full mode="ios">iOS</button>
  <button ion-button full mode="wp">Windows</button>
</div>
```

Figure 11-1 shows the screenshot of these three buttons running on the iPhone emulator. We can see that the styles of these buttons are slightly different.



Figure 11-1. Buttons with different modes

Ionic has different Sass variables (<http://ionicframework.com/docs/v2/theming/overriding-ionic-variables/>) to configure the styles. These variables can be overridden in the file `src/theme/variables.scss`. For example, the variable `$button-md-font-size` configures the button font size of mode `md`. The default value is `1.4rem`. We can add the code below to the file `variables.scss` to change the variable's value.

```
$button-md-font-size: 1.6rem;
```

Colors

The file `variables.scss` defines several variables related to colors; see Listing 11-2. The variable `$colors` is a map for named colors. These color names can be used in the property `color` for components. We can update values of these colors to change the theming.

Listing 11-2. Colors in `variables.scss`

```
$text-color:      #000;
$background-color: #fff;

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

We can also add custom names to this `$colors` map and use them in the property `color`. We can add a single value for the named color, or add a value with properties `base` and `contrast`: `base` means the background color, while `contrast` means the text color. In Listing 11-3, we add two colors: `myColor` with a single value and `anotherColor` with `base` and `contrast`.

Listing 11-3. Add custom colors

```
$colors: (
  myColor:    #f00,
  anotherColor: (
    base:      #0f0,
    contrast:  #fff
  )
);
```

In the Sass file, we can use the function `color()` to get the color value. The function `color()` takes three parameters: the first parameter is the color map, that is, `$colors`; the second parameter is the color name, that is, `myColor` or `anotherColor`; the last parameter is the color key, that is, `base` or `contrast`. The last parameter can be null. We can use `color($colors, anotherColor, base)` to get the base color of `anotherColor`.

```
div {
  background: color($colors, anotherColor, base);
}
```

Config

For each mode, Ionic has default configuration values for different properties. For example, the property `spinner` for configuring the loading spinner has the default value of `ios` on the mode `ios`, `crescent` on the mode `md` and `circles` on the mode `wp`. These configurations can be overridden in different ways.

In the class `AppModule`, we use the method `IonicModule.forRoot()` to create the module imports. This method can take a second parameter that configures the properties for different platforms. In Listing 11-4, we configure the `spinner` property to always use the value `ios`. We also configure the property `backButtonText` to an empty string only for the platform `ios`.

Listing 11-4. Override default configurations

```
@NgModule({
  imports: [
    IonicModule.forRoot(MyApp, {
      spinner: 'ios',
      platforms: {
        ios: {
          backButtonText: '',
        }
      }
    })
  ]
})
```

When testing in the browser, we can also use the URL query string to configure the properties. The query string parameters are added like `ionic<NAME>=<value>`. For example, the URL `http://localhost:8100/?ionic.Spinner=ios` configures the property `spinner` to use the value `ios`.

Configurations can also be updated programmatically using the class `Config`. The class `Config` has the method `get(key, fallbackValue)` to get a config value for a given key. If the config value was not found or the value is null, the `fallbackValue` is returned. The method `set(platform, key, value)` sets the config value for a given platform. If the platform is blank, then the config value is set to all platforms. The instance of class `Config` can be injected into components.

Storage

In the Hacker News app, we store all the user data in the Firebase database, so the user can access all the data across different devices. For some cases, it's not required that the data needs to be shared between different devices. For this kind of data, we can store the data on the device.

In this case, we can use the key/value pairs storage provided by Ionic. The package `@ionic/storage` is already installed as part of the starter template, so we can use it directly.

The storage stores key/value pairs. The value of each pair can be data of any type. If the value is a JavaScript object, it's serialized to a JSON string before saving. When the data is retrieved, the JSON string is deserialized back to a JavaScript object. The Ionic storage package wraps the `localForage` library (<https://github.com/localForage/localForage>). It provides a common API to access different storage engines, including SQLite, IndexedDB, WebSQL, and `localStorage`. The actual engine used in the runtime depends on the availability of the platform. The best storage engine to use is SQLite, because it's natively supported on iOS and Android platforms. We can install the plugin `cordova-sqlite-storage` to make SQLite available on different platforms.

```
$ cordova plugin add cordova-sqlite-storage --save
```

We use the class `Storage` from `@ionic/storage` to interact with the underlying storage engine. The class `Storage` needs to be added to the array of providers of the app's module. The instance of class `Storage` can also be injected into components. `Storage` has following methods.

- `get(key)` – Get the value by key.
- `set(key, value)` – Set the value of the given key.
- `remove(key)` – Remove the given key and its value.
- `clear()` – Clear the whole store.
- `keys()` – Get all the keys in the store.
- `length()` – Get the number of keys.
- `forEach(callback)` – Invoke the callback function for each key/value pair in the store.

Most of the operations in `Storage` are asynchronous. The return values of methods `get()`, `set()`, `remove()`, and `clear()` are all Promise objects that are resolved when the operations are completed. In Listing 11-5, we use `set()` to set the value first, then use `get()` to read the value and assign it to the property value.

Listing 11-5. Storage

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

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



```

export class StoragePage {
  value: any;
  constructor(public navCtrl: NavController,
               public navParams: NavParams,
               public storage: Storage) {}

  ionViewDidLoad() {
    const obj = {
      name: 'Alex',
      email: 'alex@example.org'
    };
    this.storage.set('value', obj)
      .then(_ => this.storage.get('value')
        .then(v => this.value = v));
  }
}

```

Push Notifications

Push notifications play an important role in mobile apps and are used in many apps. Push notifications can be used to deliver messages to the user when the app is running in the background. They also can be used to deliver data to the app to update the UI when the app is running in the foreground. Firebase offers the feature Cloud Messaging (<https://firebase.google.com/docs/cloud-messaging/>) to deliver messages to client apps.

We need to install the Cordova plugin `cordova-plugin-fcm` (<https://github.com/fechanique/cordova-plugin-fcm>) to use Firebase Cloud Messaging (FCM). The Firebase configuration files for iOS and Android, that is, the `GoogleService-info.plist` and `google-services.json`, should be downloaded and copied into the project's root directory. FCM provides three key capabilities. It can send notification messages or data messages. It allows specifying different targets for messages distributions, can be single devices, groups of devices, or devices subscribed to topics. The client apps can also send messages to the server.

```
$ cordova plugin add cordova-plugin-fcm --save
```

There are two types of messages in FCM. Notification messages are displayed by FCM when the app is running in the background. When the notification is tapped, the app is brought to the foreground and the message is handled by the app. Notification messages have a predefined structure for FCM to consume. Data messages are handled by the app. They contain custom key/value pairs with meanings specific to different apps. Data messages are handled when the app is running in the foreground. Comparing to notification

messages, the maximum size of a data message is 4KB, while the maximum size of a notification message is 2KB. If the app needs to keep updating the UI based on the server-side data, using data messages is a good choice.

We can use JSON to describe FCM messages. In Listing 11-6, the property `notification` represents the notification object. The property `data` represents the custom key/value pairs. It's possible to contain both notification and data payloads in the same message. For this kind of messages, when the app is running in the background, the notification is displayed by FCM. When the notification is tapped, the data payload is handled by the app. When the app is running in the foreground, the whole message body is handled by the app.

Listing 11-6. FCM messages in JSON format

```
{
  "to" : "",
  "notification" : {
    "body" : "You got a new message.",
    "title" : "New message",
    "icon" : "myicon"
  },
  "data" : {
    "from" : "alex",
    "content" : "Hello!"
  }
}
```

Messages can be sent using the Firebase console or FCM REST API (<https://firebase.google.com/docs/cloud-messaging/http-server-ref>). We use Android as the example to show the usage of FCM. If the app only needs to receive notification messages, then after installing the plugin `cordova-plugin-fcm`, no code changes are required. The app can already support receiving notification messages. If the app needs to receive data messages, then we need to add the code to process the message payload.

The sample scenario is to display a name that pushed to the app using notifications. In Listing 11-7, the class `NotificationPage` is the page to show the name. `FCMPlugin` is the JavaScript object provided by the plugin `cordova-plugin-fcm`. We use the method `FCMPlugin.onNotification()` to add callback handler for notifications. Because using `FCMPlugin` requires the native plugin, the call to `FCMPlugin.onNotification()` is added in the resolve callback of `Platform.ready()`. The notification callback handler receives the data as the argument. We use the data to update the value of the property `name`. The data object contains the property `wasTapped` to check if this message is received after the user tapped the notification. Here we use the method `NgZone.run()` to make sure the changes in the callback handler can be detected by Angular.

Listing 11-7. Display name using notifications

```

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

declare var window: any;

@Component({
  selector: 'page-notification',
  templateUrl: 'notification.html'
})
export class NotificationPage {
  name: string;
  constructor(public navCtrl: NavController,
              public navParams: NavParams,
              public platform: Platform,
              private zone: NgZone) {}

  ionViewDidLoad() {
    this.platform.ready().then(_ => {
      window.FCMPlugin.onNotification(data => {
        this.zone.run(() => {
          this.name = data.name;
        })
      });
    });
  }
}

```

The format of data object passed to the notification callback handler is different than the format in Listing 11-6. There is no nested properties notification or data in the object; see Listing 11-8.

Listing 11-8. The format of data object

```

{
  "body": "You got a new name!",
  "title": "New name",
  "name": "Alex",
  "wasTapped": false
}

```

Now we can test the notifications using the Firebase console. Open **Notifications** in the sidebar and click **New Message** to create a new message. In the dialog of Figure 11-2, **Message text** is the text to display in the notification. In the **Target** section, we need to specify the targets to receive notifications. In the **Advanced options** section, we can specify the message title. It's required to add the custom data for the name to display.

Message text

You got a new name!

Message label (optional) ⓘ

Enter message nickname

Delivery date ⓘ


Send Now

Target

☒ User segment ☐ Topic ☐ Single device

Target user if...


App

 com.ionicframework.demoapp478522

▼

AND

ADD TARGET

 Conversion events ⓘ

▼

Advanced options ⓘ

^

All fields optional

Title ⓘ

New name

Custom data ⓘ

name

Alex

Key

Value

Priority ⓘ

Sound

Figure 11-2. Create a new message

When the app is running in the foreground and we send a message, the UI will be updated. When the app is running in the background, a notification message is displayed. When the notification is tapped by the user, the app is brought to the foreground and the value is updated. Figure 11-3 shows the screenshot of the displayed notification.

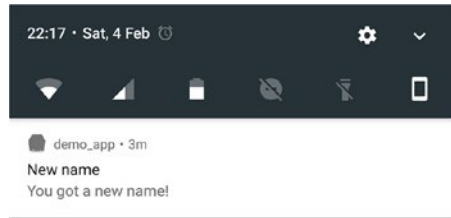


Figure 11-3. Notifications

When a message is sent to the app using the Firebase console when the app is running in the background; if the user tapped the notification, the UI may not be updated with the value in the message. This is because the message sent from Firebase console doesn't have the property "click_action": "FCM_PLUGIN_ACTIVITY" in the payload. We should use the plugin cordova-plugin-fcm test tool (<https://cordova-plugin-fcm.appspot.com/>) or the FCM REST API to send the messages.

Note On Android, the plugin cordova-plugin-fcm is currently not compatible with the plugin cordova-plugin-googleplus we used in Chapter 7 due to a version conflict of the dependent library `com.google.android.gms` introduced by the shared Gradle plugin `com.google.gms.google-services`. Only one plugin can be installed at the same time.

Summary

In this chapter, we discuss some advanced topics in Ionic 2, including Platform, theming, Config, Storage, and push notifications. These topics may not be used in every app, but can be very useful when you need to customize your app. Using push notifications is a good way to engage better with your users. In the next chapter, we'll discuss how to add end-to-end tests and continuous integration for Ionic 2 apps.

End-to-End Test and Build

In the previous chapters, we focus on the implementation of different user stories and the Ionic framework itself. In this chapter, we'll focus on some nonfunctional requirements, including end-to-end test, build, and publish.

End-to-End Test with Protractor

Protractor (<http://www.protractortest.org/>) is an end-to-end test framework for AngularJS applications. The name “Protractor” comes from the instrument that is used to measure angles, indicating its close relationship with AngularJS. Most of the end-to-end testing tools are using Selenium WebDriver (<http://www.seleniumhq.org/projects/webdriver/>) to drive a browser as a user would. You can use the WebDriver API directly, or use tools like Nightwatch.js (<http://nightwatchjs.org/>) or WebdriverIO (<http://webdriver.io/>), which wrap the WebDriver API to provide an easier-to-use API. Protractor also uses WebDriver API, but it has a close integration with AngularJS to make find elements much easier.

We use yarn to install Protractor first. This also installs dependent package `webdriver-manager` that manages the Selenium server for testing.

```
$ yarn add --dev protractor
```

Then we need to update `webdriver-manager` to install drivers for different browsers. This is done by running the command `update` of `webdriver-manager`.

Because we didn't install the package `webdriver-manager` globally, we must use the full-qualified path to access the binary of `webdriver-manager`.

```
$ node_modules/webdriver-manager/bin/webdriver-manager update
```

After the command `update` is finished, we can start the Selenium server using the command below.

```
$ node_modules/webdriver-manager/bin/webdriver-manager start
```

To make sure that `webdriver-manager` is updated, we can add a `postinstall` script to `package.json` to run `webdriver-manager update` after the packages installation.

Protractor Config

Protractor is configured using the file `protractor.conf.js` in the root directory; see Listing 12-1. This file contains different settings. Refer to this page (<http://www.protractortest.org/#/api-overview>) for more information about the config.

- `seleniumAddress` – Set the Selenium server address. `http://localhost:4444/wd/hub` is also the default Selenium address, so the property `seleniumAddress` can be omitted.
- `allScriptsTimeout` – Set the timeout settings to 15 seconds.
- `useAllAngular2AppRoots` – Set to wait for any Angular 2 apps on the page instead of just the one matching the root element.
- `specs` – Set the pattern to find all specs files. Here we use all the files with the suffix `.e2e_spec.ts` in the directory `e2e`.
- `capabilities` – Set the capabilities requirements for Selenium server. Here we require using Chrome.
- `directConnect` – Connect directly to the browser without using a Selenium server. This option can be enabled for Chrome and Firefox to speed up the execution of test scripts. When this property is set to `true`, Selenium related settings are ignored.
- `baseUrl` – Set the base URL for testing.

- framework – Set the testing framework. Jasmine is the recommended framework.
- jasmineNodeOpts – Set the options for Jasmine.
- beforeLaunch – Set the action to run before any environment setup. Here we register the directory e2e to ts-node, so ts-node compiles TypeScript code on-the-fly during the execution of the script.
- onPrepare – Set the action to run before the specs are executed. Here we configure Jasmine to add two reporters. SpecReporter is used to output testing result to console. Jasmine2HtmlReporter is used to write test results to HTML files.

Listing 12-1. protractor.conf.js

```
const SpecReporter = require('jasmine-spec-reporter');
const Jasmine2HtmlReporter = require('protractor-jasmine2-html-reporter');

exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  allScriptsTimeout: 15000,
  useAllAngular2AppRoots: true,
  specs: [
    './e2e/**/*.e2e_spec.ts'
  ],
  capabilities: {
    'browserName': 'chrome'
  },
  directConnect: true,
  baseUrl: 'http://localhost:9100/',
  framework: 'jasmine',
  jasmineNodeOpts: {
    showColors: true,
    showTiming: true,
    defaultTimeoutInterval: 30000,
  },
  beforeLaunch: function() {
    require('ts-node').register({
      project: 'e2e'
    });
  },
  onPrepare: function() {
    jasmine.getEnv().addReporter(new SpecReporter());
    jasmine.getEnv().addReporter(new Jasmine2HtmlReporter({
      savePath: './_test-output/e2e',
    }));
  }
};
```



```
    browser.ignoreSynchronization = true;
  }
};
```

The `baseUrl` we use for testing is `http://localhost:9100`, not the port 8100 of the Ionic test server. For end-to-end tests, we should use the production build. We use Python to serve the directory `www` as a static website on port 9100. We add another npm script `start_e2e` with the content `cd www && python -m SimpleHTTPServer 9100` and to start the server for testing. We add a new npm script `prod` with the content `ionic-app-scripts build --prod` to create the production build. We should run the command `yarn run prod` first to build the app, then run `yarn run start_e2e` to start the server.

In the function `onPrepare`, we also set `browser.ignoreSynchronization` to `true`. This is necessary to disable the synchronization checks in Protractor. Otherwise, all the tests will fail with timeout errors: `Timed out waiting for Protractor to synchronize with the page after 15 seconds while waiting for element with locator`. This is because Firebase uses `WebSocket` to connect to the server and the connection blocks synchronization in Protractor.

Then we can add another script in `package.json` to run Protractor. Now we can run `yarn run e2e` to start the Protractor tests.

```
{
  "scripts": {
    "e2e": "protractor"
  }
}
```

Top Stories Page Test

Now we add the first end-to-end test suite for the top stories page. The spec file is written with Jasmine, so we can leverage what we already know in writing unit tests. The first test spec verifies that there should be 10 stories. The method `browser.get('')` loads the page in the browser. Because we already configure the `baseUrl` in `protractor.conf.js`, the empty string means loading the base URL. The method `browser.sleep(5000)` makes the tests to sleep 5 seconds to allow the loading to finish. The method `element.all(by.css('item'))` finds all `<item>` elements, then we verify the number to be 10.

In the second spec, we use the method `browser.executeScript()` to execute some JavaScript code in the browser. The script `document.getElementsByTagName("ion-infinite-scroll")[0].scrollIntoView();`

makes the content to scroll to show the component `ion-infinite-scroll`, which triggers the loading of more items. We then verify the number of items should be 20.

Listing 12-2. Test spec for the top stories page

```
import { browser, element, by } from 'protractor';

describe('top stories page', () => {
  it('should show 10 stories', () => {
    browser.get('');

    browser.sleep(5000);
    const stories = element.all(by.css('item'));
    expect(stories.count()).toEqual(10);
  });

  it('should show more stories when scrolling down', () => {
    browser.get('');

    browser.driver.sleep(5000);
    let stories = element.all(by.css('item'));
    expect(stories.count()).toEqual(10);

    browser.executeScript('document.getElementsByTagName("ion-infinite-
    scroll")[0].scrollIntoView();');
    browser.sleep(5000);
    stories = element.all(by.css('item'));
    expect(stories.count()).toEqual(20);
  });
});
```

Page Objects and Suites

In the test specs of Listing 12-2, we can see a lot of code duplication. We can refactor the test code using the Page Objects pattern. A page object describes a page and encapsulates all logic related to this page. These page objects can be reused in different tests.

In Listing 12-3, `TopStoriesPage` is the class for the top stories page. The method `get()` loads the page, `scrollDown()` scrolls down to trigger the loading of more items, `getStoriesCount()` gets the number of stories in the page. In the method `beforeEach()`, an object of `TopStoriesPage` is created for each test spec and used to interact with the page.

Listing 12-3. Page object of top stories page

```
import { browser, element, by } from 'protractor';

class TopStoriesPage {
  get() {
    browser.get('');
    browser.sleep(5000);
  }

  scrollDown() {
    browser.executeScript('document.getElementsByTagName("ion-infinite-
      scroll")[0].scrollIntoView();');
    browser.sleep(5000);
  }

  getStoriesCount() {
    const stories = element.all(by.css('item'));
    return stories.count();
  }
}

describe('top stories page', () => {
  beforeEach(() => {
    this.topStoriesPage = new TopStoriesPage();
    this.topStoriesPage.get();
  });

  it('should show 10 stories', () => {
    expect(this.topStoriesPage.getStoriesCount()).toEqual(10);
  });

  it('should show more stories when scrolling down', () => {
    expect(this.topStoriesPage.getStoriesCount()).toEqual(10);

    this.topStoriesPage.scrollDown();
    expect(this.topStoriesPage.getStoriesCount()).toEqual(20);
  });
});
```

We can also group test specs into different suites. In the `protractor.conf.js`, we can use the property `suites` to configure suites and their included spec files; see Listing 12-4.

Listing 12-4. Protractor test suites

```
exports.config = {
  suites: {
    top_stories: './e2e/top_stories.e2e_spec.ts',
  },
};
```

Then we can use `--suite` to specify the suites to run.

```
$ protractor --suite top_stories
```

User Management Test

Now we add test specs for user management-related features. The class `LoginPage` in Listing 12-5 is the page object for the login page. In the method `get()`, we open the side menu and click the button to show the login modal dialog. In the method `login()`, we use `sendKeys()` to input the email and password, then click the button to log in. `$$` is short for `element(by.css())`. In the method `canLogin()`, we check the existence of the login button. In the method `isLoggedIn()`, we check the existence of the logout button. In the method `logout()`, we click the logout button to log out the current user.

Listing 12-5. Page object of login page

```
import { browser, element, by, $$ } from 'protractor';

export class LoginPage {
  get() {
    browser.get('');
    browser.sleep(5000);

    this.openMenu();

    $(' #btnShowLogin').click();
    browser.sleep(2000);
  }

  openMenu() {
    $('button[menutoggle]').click();
    browser.sleep(2000);
  }

  login(email: string = 'a@b.com', password: string = 'password') {
    $('input[name=email]').sendKeys(email);
    $('input[name=password]').sendKeys(password);
    $(' #btnLogin').click();
    browser.sleep(5000);
  }

  canLogin() {
    return element(by.css(' #btnShowLogin')).isPresent();
  }
}
```

```
isLoggedIn() {
  return element(by.css('#btnLogout')).isPresent();
}

logout() {
  this.openMenu();
  $('#btnLogout').click();
  browser.sleep(1000);
}
}
```

With the class `LoginPage`, the test suite for user management is very simple. When the page is loaded, we check the user can do the login. After we call the method `login()`, we check the user is logged in. Then we call the method `logout()` and check that the user can do the login again; see Listing 12-6.

Listing 12-6. Test spec of login page

```
import { LoginPage } from './login_page';

describe('user', () => {
  it('should be able to log in and log out', () => {
    const loginPage = new LoginPage();
    loginPage.get();
    expect(loginPage.canLogIn()).toBe(true);
    loginPage.login();
    expect(loginPage.isLoggedIn()).toBe(true);
    loginPage.logout();
    expect(loginPage.canLogIn()).toBe(true);
  });
});
```

Favorites Page Test

The test suite for the favorites page in Listing 12-7 uses the class `LoginPage` to handle the login. The class `FavoritesPage` is the page object for the favorites page. In the method `addToFavorite()`, we find the first item element with a Like button, which is the item to add to the favorites. We click the Like button to add it and return the item's title. In the method `isInFavorites()`, we go to the favorites page and check the title of the newly liked item is in the array of the titles of all items. The test spec checks that items can be added to the favorites.

Listing 12-7. Favorites page test

```

import { browser, element, by, $$ } from 'protractor';
import { LogInPage } from './login_page';

class FavoritesPage {
  logInPage: LogInPage = new LogInPage();

  get() {
    this.logInPage.get();
    this.logInPage.logIn();
  }

  openMenu() {
    $('button[menutoggle]').click();
    browser.sleep(2000);
  }

  viewTopStories() {
    this.openMenu();
    $('#btnShowTopStories').click();
    browser.sleep(5000);
  }

  viewFavorites() {
    this.openMenu();
    $('#btnShowFavorites').click();
    browser.sleep(5000);
  }

  addToFavorite() {
    this.viewTopStories();
    const itemElem = $('item').filter(elem => {
      return elem.element(by.css('.btnLike')).isPresent();
    }).first();
    if (itemElem) {
      const title = itemElem.element(by.css('h2')).getText();
      itemElem.element(by.css('.btnLike')).click();
      browser.sleep(2000);
      return title;
    }
    return null;
  }

  isInFavorites(title: string) {
    this.viewFavorites();
    expect($$('h2').map(elem => elem.getText())).toContain(title);
  }
}

```

```
describe('favorites page', () => {
  beforeEach(() => {
    this.favoritesPage = new FavoritesPage();
    this.favoritesPage.get();
  });

  it('should add stories to the favorites', () => {
    const title = this.favoritesPage.addToFavorite();
    if (!title) {
      fail('No stories can be added.');
```

Build

After we added unit tests and end-to-end tests, we need to introduce the concept of continuous integration that makes sure every commit is tested.

PhantomJS for Unit Tests

Currently we use Chrome to run unit tests, which is good for local development and debugging, but makes the continuous integration harder as it requires managing external browser processes. A better choice is to use the headless browser PhantomJS (<http://phantomjs.org/>).

Switching from Chrome to PhantomJS is an easy task; we just need to install the package `karma-phantomjs-launcher` and update the settings browsers in `karma.conf.js` to be `['PhantomJS']`.

Gitlab CI

We can integrate the app with different continuous integration servers. We use Gitlab CI as the example. If you have a local Gitlab installation, you need to install Gitlab Runner (<https://docs.gitlab.com/runner/>) to run CI jobs.

Gitlab uses Docker to run continuous integrations. We add the file `.gitlab-ci.yml` in the root directory to enable CI on Gitlab. We use the `marcoturi/ionic` as the Docker image to run. In the `before_script`, we configure the yarn cache folder and use yarn to install the dependencies. In the test stage, we start the Xvfb server for Chrome to run. Then we use yarn to run different tasks. We run the unit tests first, then run the end-to-end tests; see Listing 12-8.

Listing 12-8. .gitlab-ci.yml

```
image: marcoturi/ionic:latest

before_script:
  - yarn config set cache-folder yarn-cache
  - yarn
  - export DISPLAY=:99

cache:
  paths:
    - yarn-cache/

test:
  script:
    - /usr/bin/Xvfb :99 -screen 0 1024x768x24 -ac +extension GLX +render
    -noreset &
    - yarn run test
    - yarn run prod
    - yarn run start_e2e
  - yarn run e2e
```

Summary

In this chapter, we discuss how to use Protractor to test Ionic 2 apps and use Gitlab CI to run continuous integration. Apart from the unit tests we add in the previous chapters, end-to-end tests are also very important in testing Ionic 2 apps to make sure that different components are integrated correctly. Continuous integration is also important for the whole development life cycle that guarantees every code commit is tested. In the next chapter, we'll discuss some topics about publishing the app.

Publish

After the app has been developed and tested, it's time to publish it.

Icons and Splash Screens

Before the app can be published, we need to replace the default icons and splash screens. Ionic can generate icons and splash screens from source images to create images of various sizes for each platform. We only need to provide an image for icon and another image for splash screen, then Ionic can generate all necessary images. Source images can be .png file, .psd file from PhotoShop or .ai file from Adobe Illustrator.

For icons, the source image should be file `icon.png`, `icon.psd` or `icon.ai` in the directory `resources` of the Ionic project. The icon image should have a size of at least 192x192 px without the round corners. For splash screens, the source image should be file `splash.png`, `splash.psd` or `splash.ai` in the directory `resources`. The splash screen should have a size of at least 2732x2732 px with the image centered in the middle.

We use the command `ionic resources` to generate those resource files for icons and splash screens; see Listing 13-1.

Listing 13-1. Generate resources

```
// Icons only
$ ionic resources --icon

// Splash screens only
$ ionic resources --splash

// Both icons and splash screens
$ ionic resources
```

Generated icons and splash screens are saved to the subdirectory `ios` and `android` of the directory `resources`; see Figure 13-1.

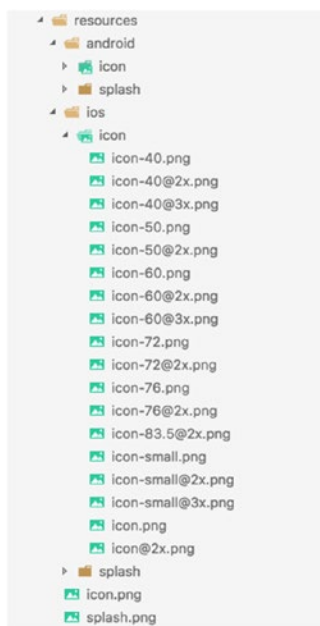


Figure 13-1. *Generated resources*

Deploy to Devices

We can deploy the app to a device for testing. For iOS, open generated project in the directory `platforms/ios` with Xcode and use Xcode to deploy to device. For Android, open generated project in the directory `platforms/android` with Android Studio to deploy to device.

Ionic CLI commands `ionic run ios` and `ionic run android` can also be used to deploy apps to the device.

View and Share with Ionic View

We can easily view and share the app with Ionic View (<http://view.ionic.io/>). Ionic View is an app created by Ionic that allows users to view Ionic apps without installing them. We can start using Ionic View with the following steps.

1. Create an account in ionic.io.
2. Use the CLI command `ionic upload` to upload the app. It prompts you to log in using your email and password if you are not logged in.
3. Install Ionic View from the app store.
4. Log into Ionic View and view the uploaded app.

After the app is uploaded to Ionic, the property `app_id` of the file `ionic.config.json` will be updated with a unique id.

You can share the app using the command `ionic share <email address>`. An email will be sent to the invited user to view the shared app. The invited user can also view the shared app in Ionic View. The app can also be shared on the Ionic apps dashboard of ionic.io.

Figure 13-2 shows the screenshot of Ionic View.

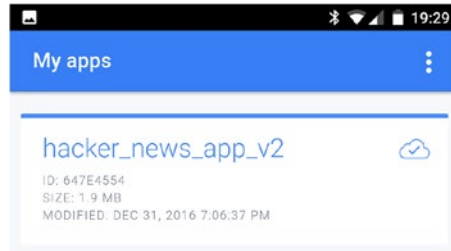


Figure 13-2. Screenshot of Ionic View

After clicking the app, we can see the action to view the app; see Figure 13-3.

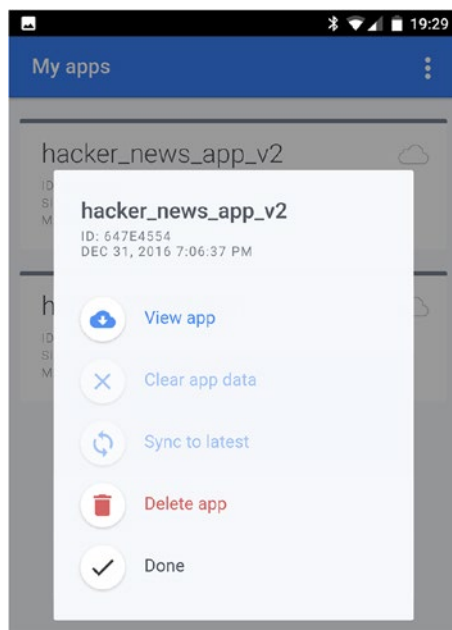


Figure 13-3. Actions of an app in Ionic View

Ionic Deploy

After we publish the app's first version to app stores, we need to continuously release new versions to the users. Usually, these new versions need to go through the same review process as the first version, which may take a long time to finish. This can delay the delivery of new features and bug fixes. For Cordova apps, since the majority code is written in HTML, JavaScript, and CSS, it's possible to perform live updates without installing new versions. These static files can be replaced by the wrapper to update to the new versions. Ionic Cloud provides the deploy service to perform live deployments.

Cloud Client

To use the Ionic Cloud services, we need to initialize the Ionic Cloud Client for the app. We need to install the package `@ionic/cloud-angular` first.

```
$ yarn add @ionic/cloud-angular
```

Then we use the command `ionic io init` to register the app in Ionic Cloud. You need to have an account in `ionic.io`. Once the app is registered, the file `ionic.config.json` is updated with generated APP ID. This id is required to configure the Cloud Client.

The configuration is done by importing the module of Cloud Client. The object `cloudSettings` in Listing 13-2 contains the configurations for Cloud Client. Here we only set the app id. The method `CloudModule.forRoot(cloudSettings)` returns the providers to import.

Listing 13-2. Configure Ionic Cloud Client

```
import { CloudSettings, CloudModule } from '@ionic/cloud-angular';
const cloudSettings: CloudSettings = {
  'core': {
    'app_id': '<APP_ID>',
  },
};

@NgModule({
  declarations: [
    ...
  ],
  imports: [
    IonicModule.forRoot(MyApp),
    CloudModule.forRoot(cloudSettings),
    AngularFireModule.initializeApp(firebaseConfig),
    FormsModule,
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    ...
  ],
  providers: [
    ...
  ]
})
export class AppModule {}
```

Deploy Service

When using the Ionic Cloud Deploy, each update is bundled as a snapshot. These snapshots can be deployed to devices. Snapshots are distributed via channels. An app can have many channels. Three default channels, production, staging, and dev are available to all apps. Custom channels can be created. An app can subscribe to any channel and receive the snapshots pushed to this channel. The channel production is used by default.

In the app, we need to add code to check for new snapshots and deploy them. This can be done using the class `Deploy` provided by the Cloud Client. Listing 13-3 shows the `DeployService` for deployment. In the method `checkForUpdates()`, we use `check()` of class `Deploy` to check for any new snapshots. When there is any snapshot, we call the method `confirmUpdate()` to show an alert to notify the user. If the user wants to update, we call the method `install()` to install the latest snapshot. In the method `install()`, we chain three actions together. The method `download()` downloads the snapshot to the device; `extract()` extracts the downloaded snapshot, and makes it as the active deployment; `load()` reloads the app to use the latest snapshot.

Listing 13-3. DeployService

```
import { Injectable } from '@angular/core';
import { Deploy } from '@ionic/cloud-angular';
import { AlertController } from 'ionic-angular';

@Injectable()
export class DeployService {
  constructor(private deploy: Deploy,
              private alertCtrl: AlertController) {

  }

  checkForUpdates() {
    this.deploy.check().then(hasUpdate => {
      if (hasUpdate) {
        this.confirmUpdate();
      }
    });
  }

  install() {
    this.deploy.download()
      .then(() => this.deploy.extract()
        .then(() => this.deploy.load()));
  }

  private confirmUpdate() {
    const confirm = this.alertCtrl.create({
      title: 'New version available',
      message: 'A new version is available. Do you want to update and reload the app?',
      buttons: [
```

```

    {
      text: 'No',
      handler: () => {
        confirm.dismiss();
      }
    },
    {
      text: 'Update & Reload',
      handler: () => {
        confirm.dismiss();
        this.install();
      }
    }
  ]
});
confirm.present();
}
}

```

After a new version is ready to deploy, we can use Ionic CLI to upload this new version to Ionic Cloud and deploy to a channel; see Listing 13-4. The first command only uploads the new version; the second command also deploys this new version to a channel. If the new version is uploaded only, then it can be deployed later from Ionic Cloud.

Listing 13-4. Upload and deploy new snapshots

```

$ ionic upload --note "bug fixes"

$ ionic upload --note "bug fixes" --deploy production

```

Not all updates can be deployed using Ionic Cloud. If the native platform is updated, or a Cordova plugin is installed or updated, we need to update the binary. These binary versions need to go through the normal app store review process. When deploying a snapshot, we can specify the binary version requirement for this snapshot. For example, after a new Cordova plugin is installed, new snapshots cannot be deployed to apps with old binary versions as the new plugin doesn't exist on those old versions.

Figure 13-4 shows the alert when a new version is available for update.

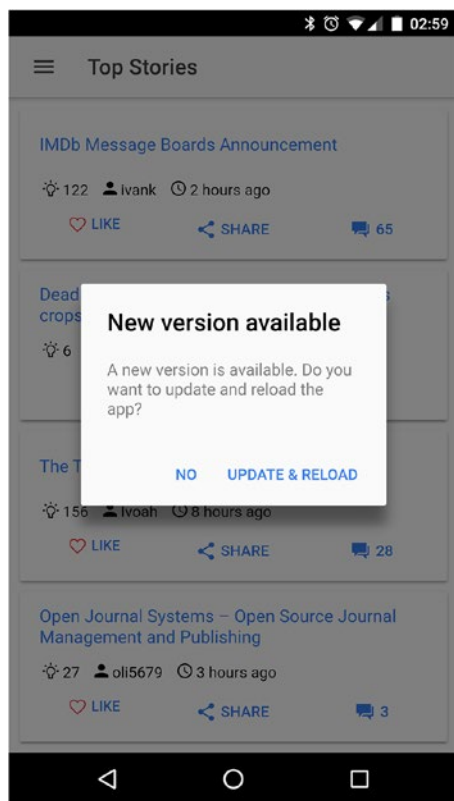


Figure 13-4. Alert for new versions

Summary

In the last chapter of this book, we discuss some important topics about publishing the app to app stores, including adding icons and splash screens and using Ionic Deploy to perform live updates to published apps.

In this book, we use the Hacker News app as the example to demonstrate the whole life cycle of building mobile apps with Ionic 2 and Firebase. We start from the introduction of hybrid mobile apps and programming languages, frameworks, libraries, and tools used in the development, then we implement different user stories. During the implementation, we discuss details about Ionic 2 components and Firebase. We also cover some advanced topics of Ionic 2. At last, we discuss some nonfunctional topics, including end-to-end tests, continuous integration, and app publish. After reading this book, you should be able to build your own mobile apps with Ionic 2 and Firebase.

Index

A

- AbstractItemService, 129–130
- AbstractItemsPage, 131
- Action sheet, 203–205
 - return value of create(), 204
 - properties, 203–204
- ActionSheetController, 203
- addButton() method, 122
- addInput() method, 121
- AlertController, 121
- Android, 165
 - Chrome DevTools for remote debugging, 15
 - Genymotion, 9
 - local development
 - environment, 7–8
 - test on emulators, 16–17
- Angular 2–34
 - bootstrapping, 40
 - components, 36–37
 - dependency injection, 38
 - class providers, 38
 - factory providers, 39
 - services in
 - component, 40
 - tokens, 39
 - value providers, 38
 - metadata, 40
 - modules, 35–36
 - services, 37
- Apache Cordova, 2–3
- AuthService, 150–154
- AuthServiceMock, 189

B

- Build
 - Gitlab CI, 236
 - PhantomJS for unit tests, 236

C

- Card layout, 195–196
- Chrome DevTools, remote debugging, 15
- Classes in TypeScript, 30–31
- Cloud Client, 242–243
- Colors, 218
 - add custom, 218
- CommentComponent, 133–134
- CommentsComponent, 134
- CommentService, 131
- Common mocks, 190–191
- Configurations, 219
 - override default, 219
- config.xml, 49
- Continuous integration, 237
- Cordova app, 48
- Cordova files
 - hooks, 50
 - platforms, 50
 - plugins, 51
 - www directory, 51

D

- Data messages, 221
- Decorators, TypeScript, 31
 - class, 31–32

Decorators, TypeScript (*cont.*)

method, 32–33

property, 33–34

Deploy service, 243–246

Deploy to devices, 240

Destructive buttons, 203

E

.editorconfig, 50

Email/password login

AuthService, 150–154

model class, 150

Empty list, 65–66

Emulators, test, 15

Android, 16–17

iOS, 15

End-to-end test, 66, 227

favorites page test, 234–236

page objects and suites, 231–232

with Protractor, 227–228

configurations, 228–230

top stories page test, 230–231

user management test, 233–234

Error handling

observable object, 112

testing, 114–115

toast, 113–114

top stories page, 115

F

Facebook login, 168–170

FavoriteServiceMock, 187–189

Favorites items, 184

addToFavorite and

removeFromFavorite, 185

screenshot of top stories page, 187

template of ItemComponent, 186

template of ItemsComponent, 186

updated ItemComponent, 185–186

Favorites page, 182–183

AbstractFavoriteItemsPage,

180–182

addToFavorite() method, 181

mergeFavorites() method, 181

removeFromFavorite()

method, 181

test/testing, 187–191, 234–236

TopStoriesPage, 180, 183–184

FavoritesPage test suite, 191–193

Favorites service, 177, 179–180

add() method, 178

favorites data, 177–178

getFavoriteItem() method, 179

getItemIds() method, 178

map operation, 178

pluck operation, 178

remove() method, 178

sortBy operation, 178

FCMPlugin, 222

Firebase, 4, 138, 193

database structure, 57, 82–83

JavaScript SDK

Angular 2 apps, 83

DataSnapshot, 85

event types, 86

pushing data, 87

reading data, 84

remove event listeners, 85

setup, 83

writing data, 86–87

navigation, 89

query data

filtering, 88–89

sorting, 88

Firebase Cloud Messaging

(FCM), 221

data, 221

display name using

notifications, 223

FCMPlugin, 222

JSON, 222

NgZone.run() method, 222

notification, 221

Flappy Bird, 2

Function type ViewComment, 134

G

Genymotion, 9, 16
 GitHub login, 170–174
 Gitlab CI, 236–237
 Google login, 164
 Grid layout, 196, 198

H

Hacker News API
 advanced list
 infinite scrolling and
 pull-to-refresh, 102
 ion-refresher and
 ion-infinite-scroll, 102–103
 updated TopStories, 103–105
 AngularFire2, 90–91
 customization
 ion-infinite-scroll, 106
 ion-refresher, 106–107
 ItemService, 92–93
 JSON content of a story, 92
 Observable.combineLatest()
 method, 97
 pagination and refresh
 button, 98–99
 updated TopStories, 100–102
 selector function, 97
 testing, scrolling and
 refresh, 107–108
 Top stories page with load
 button, 98
 updated item.html, 94
 updated ItemService, 95–97
 updated items.html, 94
 updated model Items, 93
 Hacker News app, 52
 API (see Hacker News API)
 assets, 51
 components, 52
 .config files
 config.xml, 49
 .editorconfig, 50
 ionic.config.json, 50

 package.json, 48
 tsconfig.json, 49
 tslint.json, 50
 declarations.d.ts, 51
 index.html, 51
 Ionic app, 48
 iOS and Android platforms, 48
 manifest.json and service-
 worker.js, 51
 pages, 52
 requirements, 47–48
 theme, 52
 Hybrid mobile apps, 1–2
 Apache Cordova, 3
 Firebase, 4
 Ionic framework, 3–4

I

Icons and splash screens, 239–240
 IDEs and editors, 9
 Inappbrowser plugin, Cordova, 117
 Alert, 121
 better solution, 122–124
 installation, 118
 opening URL, 118–120
 events, 120
 styles, 125
 testing, 126
 Input decorator factory, 63
 Interfaces in TypeScript, 28
 classes contracts, 29
 shape of values, 28–29
 Ionic CLI, 5, 20
 Ionic components
 action sheet, 203–205
 popover, 206–208
 slides, 209–211
 tabs, 212–214
 ionic.config.json, 50
 Ionic deploy, 242
 Ionic framework, 3–4, 47
 IonicModule.forRoot() method, 52, 219
 Ionic view, 241–242
 iOS app, 7, 15, 164

Item component, 63–65
Item model, 58
Items component, 65
Items loading service, 76–77

J

Jasmine, 45

K

Karma, 45

L

Labels, 146
List component
 avatars, 61
 grouping of items, 59–60
 header and separators, 59
 icons, 60
 simple list, 58–59
 thumbnails, 61–62
Loading
 emit only one single value, 110
 LoadingController, 109
 testing
 create Jasmine spy
 objects, 112
 ItemServiceMock to emit
 loading event, 111
 loadingControllerStub
 object, 112
 updated ItemService, 108–109
 updated TopStories, 109–110
Local development environment, 5
 Android, 7–9
 IDEs and editors, 9
 Ionic CLI, 5
 iOS, 7
 Node.js, 5
 yarn, 6
Logger service, 38
Login, 165–168
Login page, 158–160

M

Menu
 AuthService, 160
 menuClose, 149
 MenuController, 149
 menuToggle, 149
 user's status, 161
Mobile apps
 development, 1
 hybrid (see Hybrid mobile apps)
 refresher, 1–2
 stores, statistics, 1
Modals, 146
Model, 128

N

NavController, 127–128
Navigation, 127
 basic usage, 127
 page, 128
Node.js, 5
Notification messages, 221

O

Object-oriented programming, 30
OpenPageService, 119–120, 122–124

P, Q

package.json, 48
Page navigation, 128
Page objects
 of login page, 233
 and suites, 231–232
PhantomJS for unit test, 236
PhoneGap, 3
Platform, 215
 dir(), 215
 is(platformName), 215
 isPortrait() and isLandscape(), 216
 isRTL(), 216
 lang(), 216
 platforms(), 215

- ready(), 216
- registerBackButtonAction
 - (callback, priority), 216
- setDir(dir), 216
- setLang(language), 216
- versions(), 215
- width() and height(), 216
- Plugin, 200
- Popover, 206
 - return value of create(), 206
 - dismiss() method, 206
 - font size chooser, 208
 - FontSizeChooserComponent, 207–208
 - template, 208
 - parameters, 206
 - PopoverPage, 206–207
 - template, 207
- Product development, 57
- Protractor, end-to-end test
 - with, 227–228
 - configurations, 228–230
 - Protractor test suites, 232
- Push notifications, 221–225

R

- Radio buttons, 141–142
- Range sliders, 144–146
- Refactored ItemService, 130–131
- Refactoring, 129
 - pages, 131–132
 - services, 129–131
- RxJS, 41
 - observables, 41
 - observers, 42
 - operators, 43
 - subject, 42

S

- Sass, 43
 - code, 125
 - mixins, 44–45
 - nesting, 44
 - variables, 43

- Selenium WebDriver, 227
- Sharing, 198–199
- Sign-up form, 154–158
- Skeleton code
 - app.component.ts, 53–54
 - app.html, 54
 - app.module.ts, 52–53
 - app.scss, 55
 - blank app, 9–10
 - Chrome, 14
 - Chrome DevTools, 15
 - local development, 13
 - main.ts, 55
 - Page1 and Page2 files, 55
 - sidemenu app, 11–12
 - tabbed app, 10
 - tutorial app, 12–13
- Slides component, 209, 211
 - ion-slides, 209
 - SlidesControl and
 - SlidesPage, 210–211
 - template of slides-control, 210
 - transitions, 209
- Storage, 219
 - components, 220

T

- Tabs, 212, 214
 - color and mode, 212
 - ion-tabs, 212
 - methods to interact, 212
 - NavController, 212–213
 - NavParams, 213
 - template, 213
- Testing, list component
 - configuration
 - angular-cli.json, 67–68
 - karma.conf.js, 68–70
 - karma-test-shim.ts, 70
 - items component
 - items.spec.ts, 71–72
 - mocks, 74
 - TestUtils, 72–73
- Karma debug UI, 76

- Testing, list component (*cont.*)
 - run tests, 75
 - tools, 66–67
- Theming, 216
 - modes for ion-button components, 217
- Third-party login, 163
- TimeAgo pipe, 64
- Timestamp, 177
- Toggles, 144
- Toolbar, 147–148
- Top stories page (TopStoriesPage), 180, 183–184, 231
 - ItemsComponent and ItemService, 78
 - ngOnInit() method, 78
 - testing, 230
 - ItemServiceMock, 80
 - test spec, 231
 - test suite, 80–81
 - top-stories.spec.ts, 81
 - top-stories.html, 79
 - top-stories.ts, 78–79
- tsconfig.json, 49
- tslint.json, 50
- TypeScript, 20
 - Angular 2–34
 - bootstrapping, 40
 - components, 36–37
 - dependency injection, 38–40
 - metadata, 40
 - modules, 35–36
 - services, 37
 - classes, 30–31
 - compile-time type checks, 20
 - decorators, 31
 - class, 31–32
 - method, 32–33
 - property, 33–34
 - functions, 25–26
 - arguments, 26–27
 - IDE support, 21
 - interfaces, 28
 - classes contracts, 29
 - shape of values, 28–29

- rich feature sets, 21
- types
 - any, 24
 - array, 23
 - Boolean, 21
 - enum, 23
 - never, 25
 - null and undefined, 22–23
 - numbers, 22
 - string, 22
 - tuple, 23
 - union, 25
 - void, 25

■ U

- Unit tests, 66
- Updated TopStoriesPage, 132
- Update user's name, 162
- User management
 - ion-checkbox, 140
 - ion-input, 140
 - ion-menu, 148
 - ion-select, 142–144
 - labels, 146
 - modals, 146
 - radio buttons, 141–142
 - range sliders, 144–146
 - test, 233–234
 - toggles, 144
 - toolbar, 147–148
 - user authentication, 139

■ V, W, X

- View comments, 133
 - CommentComponent, 133–134
 - CommentsComponent, 134
 - CommentsPage, 136–137
 - items, 135
 - viewComment(), 135

■ Y, Z

- Yarn, 6